**Set 1**

1. import re

# Sample text containing emails

text = """

Hello, please contact us at support@example.com for further information.

You can also reach out to john\_doe123@mail.example.org or jane.doe@company.co.uk.

Invalid emails like example@.com or user@@domain.com should not be captured.

"""

# Regular expression pattern for a valid email address

email\_pattern = r'[a-zA-Z0-9\_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+'

# Function to extract and validate email addresses

def extract\_emails(text):

# Find all matches of the pattern in the text

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

return emails

# Call the function and display the extracted emails

valid\_emails = extract\_emails(text)

print("Valid email addresses found:")

for email in valid\_emails:

print(email)

2.

# Import necessary libraries

from sklearn.feature\_extraction.text import CountVectorizer

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

from sklearn.naive\_bayes import MultinomialNB

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

# Sample dataset: messages and their labels (1 = spam, 0 = ham)

data = [

("Free entry in 2 a wkly comp to win FA Cup final tickets", 1),

("Congrats! You've won a $1,000 Walmart gift card. Call now!", 1),

("Hey, are we still meeting tomorrow?", 0),

("URGENT! Your mobile number has won $5000", 1),

("Can you send me the report by tomorrow?", 0),

("Call your mom. She’s asking about you.", 0),

("You have been selected to receive a free laptop", 1),

("Let's have a call to discuss the project", 0)

]

# Split the data into messages and labels

messages = [item[0] for item in data]

labels = [item[1] for item in data]

# Step 1: Convert the text data into numerical form using CountVectorizer

vectorizer = CountVectorizer()

X = vectorizer.fit\_transform(messages) # Convert text to a matrix of token counts

# Step 2: Split the data into training and testing sets (80% training, 20% testing)

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

# Step 3: Train a Naive Bayes classifier (MultinomialNB)

nb\_classifier = MultinomialNB()

nb\_classifier.fit(X\_train, y\_train)

# Step 4: Make predictions on the test set

y\_pred = nb\_classifier.predict(X\_test)

# Step 5: Evaluate the model's performance

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

print(f"Accuracy: {accuracy:.2f}")

# Print a classification report (precision, recall, F1-score)

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred, target\_names=["Ham", "Spam"]))

3

from sklearn.feature\_extraction.text import TfidfVectorizer

import pandas as pd

# Sample sentences

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."

]

# Step 1: Initialize the TfidfVectorizer

vectorizer = TfidfVectorizer(stop\_words='english', max\_df=0.85)

# Step 2: Fit and transform the sentences to create a TF-IDF matrix

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

# Step 3: Extract feature names (keywords/phrases)

feature\_names = vectorizer.get\_feature\_names\_out()

# Step 4: Convert the matrix into a pandas DataFrame for better readability

df = pd.DataFrame(tfidf\_matrix.T.toarray(), index=feature\_names)

df.columns = [f"Sentence {i+1}" for i in range(len(sentences))]

# Step 5: Display the DataFrame

print("TF-IDF Scores for Each Word in Each Sentence:")

print(df)

# Step 6: Identify and extract key phrases (top keywords for each sentence)

for i in range(len(sentences)):

print(f"\nTop keywords in Sentence {i+1}:")

sorted\_keywords = df[f"Sentence {i+1}"].sort\_values(ascending=False)

print(sorted\_keywords.head(5)) # Extract the top 5 keywords for each sentence

4

import nltk

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

# Function for Named Entity Recognition (NER)

def named\_entity\_recognition(text):

# Tokenize the text into words

words = word\_tokenize(text)

# Apply part-of-speech tagging

pos\_tags = pos\_tag(words)

# Apply Named Entity Recognition (NER)

named\_entities\_tree = ne\_chunk(pos\_tags)

# Extract named entities

named\_entities = []

for subtree in named\_entities\_tree:

if isinstance(subtree, nltk.Tree):

entity\_name = " ".join([token for token, pos in subtree.leaves()])

entity\_type = subtree.label()

named\_entities.append((entity\_name, entity\_type))

return named\_entities

# Function for Reference Resolution (basic example)

def reference\_resolution(text):

# Basic reference resolution example: simple pronoun substitution

# For a more advanced system, libraries like neuralcoref or spacy could be used

resolved\_text = text.replace("it", "Harvard University") # A basic example

return resolved\_text

# Example text for Named Entity Recognition

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

# Named Entity Recognition

entities = named\_entity\_recognition(text\_ner)

print("Named Entities:")

for entity, entity\_type in entities:

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

# Example text for Reference Resolution

text\_ref = "Harvard University, located in Cambridge, Massachusetts, is a prestigious institution. It has a rich history."

# Reference Resolution

resolved\_text = reference\_resolution(text\_ref)

print("\nResolved Text:")

print(resolved\_text)

**set-2**

1

def is\_equal\_zeros\_ones(s):

"""

This function simulates a finite state automaton to check if a string

has an equal number of '0's and '1's.

"""

# Counter to keep track of the difference between 0's and 1's

count = 0

# Traverse through the string

for char in s:

if char == '0':

count += 1 # Increment count for '0'

elif char == '1':

count -= 1 # Decrement count for '1'

else:

return False # If any other character is found, return False

# If the count is 0, it means equal number of '0's and '1's

return count == 0

# Test cases

test\_strings = [

"0011", # Equal number of 0's and 1's

"1010", # Equal number of 0's and 1's

"0101", # Equal number of 0's and 1's

"110", # Not equal

"111000", # Equal number of 0's and 1's

"10001", # Not equal

"011001" # Not equal

]

# Check each string and print the result

for s in test\_strings:

result = is\_equal\_zeros\_ones(s)

print(f"String '{s}' has equal 0's and 1's: {result}")

2.

import spacy

# Load the pre-trained small English model in spaCy

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

def extract\_named\_entities(text):

# Process the text with spaCy NLP model

doc = nlp(text)

# Extract and print named entities with their labels

for ent in doc.ents:

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

# Input sentence

text = "Barack Obama was the 44th President of the United States, and he was born in Honolulu, Hawaii."

# Perform Named Entity Recognition

extract\_named\_entities(text)

3

import spacy

import wikipediaapi

# Initialize spaCy NLP model

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

# Initialize Wikipedia API

wiki\_wiki = wikipediaapi.Wikipedia('en')

def resolve\_entity\_to\_wikipedia(entity):

# Try to get a Wikipedia page for the entity

page = wiki\_wiki.page(entity)

if page.exists():

return page.title

return None

def process\_text(text):

# Process the text with spaCy NLP model

doc = nlp(text)

# Extract and resolve named entities

for ent in doc.ents:

resolved\_entity = resolve\_entity\_to\_wikipedia(ent.text)

if resolved\_entity:

print(f"Entity: {ent.text} -> Wikipedia Page: {resolved\_entity}")

else:

print(f"Entity: {ent.text} -> No Wikipedia Page found")

# Input sentences

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."

]

# Process each sentence

for sentence in sentences:

print(f"\nProcessing sentence: '{sentence}'")

process\_text(sentence)

4

import spacy

# Load spaCy's small English model

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

def calculate\_similarity(sent1, sent2):

# Process sentences with spaCy

doc1 = nlp(sent1)

doc2 = nlp(sent2)

# Compute the similarity between the sentences

similarity = doc1.similarity(doc2)

return similarity

def evaluate\_coherence(text):

# Split text into sentences

sentences = text.strip().split('\n')

# Calculate similarities between consecutive sentences

similarities = []

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

sim = calculate\_similarity(sentences[i], sentences[i + 1])

similarities.append(sim)

print(f"Similarity between '{sentences[i]}' and '{sentences[i + 1]}': {sim:.2f}")

# Average similarity score as an indication of coherence

average\_similarity = sum(similarities) / len(similarities) if similarities else 0

print(f"Average Similarity Score: {average\_similarity:.2f}")

# Sample text

sample\_text = """

Once upon a time, there was a young boy named Peter.

He lived in a small village.

One day, he decided to explore the nearby forest.

"""

# Evaluate coherence of the sample text

evaluate\_coherence(sample\_text)

**set 3**

1. import nltk

from nltk.tokenize import word\_tokenize

from nltk.stem import WordNetLemmatizer

from nltk.corpus import wordnet

from nltk import pos\_tag

# Ensure necessary NLTK resources are downloaded

nltk.download('punkt')

nltk.download('wordnet')

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

# Initialize the lemmatizer

lemmatizer = WordNetLemmatizer()

def get\_wordnet\_pos(treebank\_tag):

"""

Convert treebank tags to wordnet tags.

"""

if treebank\_tag.startswith('J'):

return wordnet.ADJ

elif treebank\_tag.startswith('V'):

return wordnet.VERB

elif treebank\_tag.startswith('N'):

return wordnet.NOUN

elif treebank\_tag.startswith('R'):

return wordnet.ADV

else:

return None

def perform\_morphological\_analysis(sentence):

# Tokenize the sentence into words

tokens = word\_tokenize(sentence)

# Get part of speech tags

pos\_tags = pos\_tag(tokens)

# Perform lemmatization

lemmatized\_words = []

for word, tag in pos\_tags:

wordnet\_pos = get\_wordnet\_pos(tag)

if wordnet\_pos:

lemmatized\_word = lemmatizer.lemmatize(word, pos=wordnet\_pos)

else:

lemmatized\_word = lemmatizer.lemmatize(word)

lemmatized\_words.append((word, tag, lemmatized\_word))

return lemmatized\_words

# Input sentence

sentence = "Unhappily, she ran quickly"

# Perform morphological analysis

result = perform\_morphological\_analysis(sentence)

# Print the results

print("Word | POS Tag | Lemmatized Form")

print("-" \* 40)

for original\_word, pos\_tag, lemmatized\_word in result:

print(f"{original\_word: <10} | {pos\_tag: <8} | {lemmatized\_word}")

2

from textblob import TextBlob

def analyze\_sentiment(text):

# Create a TextBlob object

blob = TextBlob(text)

# Get the polarity score

polarity = blob.sentiment.polarity

# Determine sentiment category

if polarity > 0:

sentiment = 'Positive'

elif polarity < 0:

sentiment = 'Negative'

else:

sentiment = 'Neutral'

return sentiment

# Sentences for sentiment analysis

sentences = [

"I love this product! It's amazing.",

"The weather is terrible today."

]

# Analyze and print the sentiment of each sentence

for sentence in sentences:

sentiment = analyze\_sentiment(sentence)

print(f"Sentence: '{sentence}'")

print(f"Sentiment: {sentiment}")

print()

3

import nltk

from nltk.corpus import wordnet

# Ensure necessary NLTK data is downloaded

nltk.download('wordnet')

nltk.download('omw-1.4')

def get\_word\_info(word):

# Retrieve synsets for the word

synsets = wordnet.synsets(word)

if not synsets:

return f"No information found for '{word}'"

# Collect information

info = []

for synset in synsets:

# Get the definition and examples

definition = synset.definition()

examples = synset.examples()

# Get synonyms

synonyms = synset.lemmas()

synonym\_words = [lemma.name() for lemma in synonyms]

info.append({

'synset': synset.name(),

'definition': definition,

'examples': examples,

'synonyms': list(set(synonym\_words)) # Remove duplicates

})

return info

def print\_word\_info(word):

info = get\_word\_info(word)

if isinstance(info, str):

print(info)

else:

for entry in info:

print(f"Synset: {entry['synset']}")

print(f"Definition: {entry['definition']}")

print(f"Examples: {', '.join(entry['examples']) if entry['examples'] else 'No examples available'}")

print(f"Synonyms: {', '.join(entry['synonyms'])}")

print()

# Sentences to analyze

sentences = [

"The cat sat on the mat.",

"The dog barked loudly.",

"She played the piano beautifully."

]

# Tokenize sentences into words and retrieve WordNet information

for sentence in sentences:

print(f"Processing sentence: '{sentence}'")

words = set(word.lower().strip('.,') for word in sentence.split()) # Simple tokenization and case normalization

for word in words:

print(f"Word: '{word}'")

print\_word\_info(word)

print("\n" + "-"\*40 + "\n")

4

import re

# Define dialog act categories and their patterns

dialog\_act\_patterns = {

'Greeting': r'\b(hi|hello|good morning|good evening|hey)\b',

'Question': r'\b(who|what|where|when|why|how|could|can|would|should|is|are|do|does)\b',

'Statement': r'\b(is|are|was|were|has|have|had|will|would|shall|might|could|should|can)\b',

'Request': r'\b(please|could|would|can|may)\b',

'Answer': r'\b(yes|no|sure|okay|alright)\b'

}

def classify\_dialog\_act(sentence):

"""

Classify the dialog act of a given sentence based on predefined patterns.

"""

# Convert sentence to lowercase for pattern matching

sentence\_lower = sentence.lower()

for act, pattern in dialog\_act\_patterns.items():

if re.search(pattern, sentence\_lower):

return act

return 'Unknown'

# Sample dialog

dialog = [

"Good morning! How's the weather today?",

"I heard it's going to be sunny and warm.",

"Could you please send me the report by 3 PM?",

"Of course, I'll send it over before the deadline.",

"Do you know where the nearest post office is?",

"The post office is two blocks down the street."

]

# Classify each sentence in the dialog

classified\_dialog = [(sentence, classify\_dialog\_act(sentence)) for sentence in dialog]

# Print results

print("Sentence | Dialog Act")

print("-" \* 30)

for sentence, dialog\_act in classified\_dialog:

print(f"{sentence: <30} | {dialog\_act}")

**set 4**

1

import re

# Sample verb conjugation rules for simple past tense

regular\_verbs\_rules = {

'walk': 'walked',

'jump': 'jumped',

'play': 'played',

'talk': 'talked',

'work': 'worked'

}

# Helper function to convert a verb to its past tense form

def get\_past\_tense(verb):

# Check if the verb is in our regular verbs list

if verb in regular\_verbs\_rules:

return regular\_verbs\_rules[verb]

# Apply simple regular verb rules if not in our predefined list

if verb.endswith('e'):

return verb + 'd'

elif verb.endswith('y'):

return verb[:-1] + 'ied'

else:

return verb + 'ed'

# Finite-State Machine to parse and generate past tense forms

def process\_sentence(sentence):

# Tokenize the sentence and identify verbs

words = sentence.lower().split()

past\_tense\_verbs = []

for word in words:

# Check if the word is a known verb or a regular verb

past\_tense\_form = get\_past\_tense(word)

# Check if the verb is different from the original (indicating a transformation)

if past\_tense\_form != word:

past\_tense\_verbs.append((word, past\_tense\_form))

return past\_tense\_verbs

# Example sentences

sentences = [

"She walked to the park yesterday",

"He jumped over the fence"

]

# Process each sentence and print the results

for sentence in sentences:

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

past\_tense\_verbs = process\_sentence(sentence)

for original, past\_tense in past\_tense\_verbs:

print(f"Verb: '{original}' -> Past Tense: '{past\_tense}'")

print("\n" + "-"\*40 + "\n")

2

from gensim.summarization import summarize

# Input document

document = """

Natural language processing (NLP) is a subfield of artificial intelligence (AI) that focuses on the interaction between computers and humans through natural language. NLP technologies are used to process, analyze, and understand large amounts of natural language data.

One of the primary applications of NLP is sentiment analysis, which determines the sentiment or emotional tone of a piece of text. Sentiment analysis is widely used in social media monitoring, customer feedback analysis, and brand reputation management.

Text summarization is another important NLP task. Extractive summarization involves selecting a subset of sentences from a text to create a shorter version that retains the most critical information. Abstractive summarization, on the other hand, generates a summary by paraphrasing and rephrasing the original text.

"""

# Perform extractive summarization

summary = summarize(document, ratio=0.5) # ratio=0.5 means the summary will be approximately 50% of the original length

# Print the summary

print("Extractive Summary:")

print(summary)

3

import re

# Define a function to evaluate logical expressions

def evaluate\_expression(expression, values):

# Replace variables in the expression with their values

for var, val in values.items():

expression = expression.replace(var, str(val))

# Evaluate the expression using Python's eval function with logical operations

# Replace logical operators with Python equivalents

expression = expression.replace('and', 'and').replace('or', 'or').replace('not', 'not')

# Use eval to evaluate the expression

try:

result = eval(expression)

except Exception as e:

print(f"Error evaluating expression '{expression}': {e}")

return None

return result

# Function to parse and evaluate a list of expressions

def parse\_and\_evaluate(expressions, values):

results = {}

for expr in expressions:

print(f"Evaluating expression: '{expr}'")

result = evaluate\_expression(expr, values)

results[expr] = result

print(f"Result: {result}")

print("\n" + "-"\*40 + "\n")

return results

# Define logical expressions

expressions = [

"x and y",

"x or (not y)",

"x and (y or (not x))"

]

# Define variable values

values = {

'x': True,

'y': False

}

# Parse and evaluate the expressions

parse\_and\_evaluate(expressions, values)

4

import openai

# Replace with your OpenAI API key

openai.api\_key = 'YOUR\_API\_KEY\_HERE'

def generate\_text(prompt):

"""

Generate text based on the given prompt using GPT-3.

"""

try:

# Make a request to the OpenAI API

response = openai.Completion.create(

engine="text-davinci-003", # You can use different models such as text-curie-001, text-babbage-001, etc.

prompt=prompt,

max\_tokens=150, # Maximum number of tokens to generate

n=1, # Number of completions to generate

stop=None, # Stop sequences, if any

temperature=0.7 # Sampling temperature (0.7 is often used for creativity)

)

# Extract the generated text

text = response.choices[0].text.strip()

return text

except Exception as e:

return f"An error occurred: {e}"

def main():

print("Welcome to the GPT-3 text generator!")

while True:

prompt = input("Enter your prompt (type 'exit' to quit): ")

if prompt.lower() == 'exit':

break

# Generate text based on the prompt

generated\_text = generate\_text(prompt)

print("\nGenerated Text:")

print(generated\_text)

print("\n" + "-"\*50 + "\n")

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

main()

**set 5**

1

import nltk

from nltk.stem import PorterStemmer

# Ensure necessary NLTK data is downloaded

nltk.download('punkt')

# Initialize the Porter Stemmer

stemmer = PorterStemmer()

# List of words to be stemmed

words = ["jumps", "jumping", "jumper", "jumped", "easily", "running", "flies", "flying", "flies"]

# Apply stemming to each word

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

# Display before and after stemming results

print("Word - Stemmed Word")

print("--------------------")

for original, stemmed in zip(words, stemmed\_words):

print(f"{original} - {stemmed}")

2

import spacy

# Load the spaCy model

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

# Function to perform dependency parsing

def dependency\_parse(sentence):

# Process the sentence

doc = nlp(sentence)

# Extract and print dependency relations

print(f"\nDependency Parsing for Sentence: '{sentence}'\n")

print(f"{'Token':<15} {'Lemma':<10} {'POS':<5} {'Tag':<10} {'Head':<15} {'Dep':<15}")

print("-" \* 65)

for token in doc:

print(f"{token.text:<15} {token.lemma\_:<10} {token.pos\_:<5} {token.tag\_:<10} {token.head.text:<15} {token.dep\_:<15}")

# Sentences to analyze

sentences = [

"The cat sat on the mat.",

"She quickly ran to the bus."

]

# Parse and display dependency relations for each sentence

for sentence in sentences:

dependency\_parse(sentence)

3

import re

# Define the text to search

text = "Regular expressions are a powerful tool for pattern matching in text. The regex is the most flexible way to search and manipulate strings."

# Define the regex pattern

pattern = r"powerful|flexible|manipulate"

# Perform regex search

matches = re.findall(pattern, text)

# Print the results

print("Text:")

print(text)

print("\nRegex Pattern:")

print(pattern)

print("\nMatches found:")

print(matches)

4

import spacy

# Load the spaCy model

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

# Define the text to analyze

text = "Microsoft Corporation is headquartered in Redmond, Washington."

# Process the text using spaCy

doc = nlp(text)

# Print the text

print("Text:")

print(text)

# Print named entities

print("\nNamed Entities, Phrases, and Concepts:")

for ent in doc.ents:

print(f"{ent.text} ({ent.label\_})")

# Print tokens and their dependencies

print("\nTokens and their dependencies:")

print(f"{'Token':<20} {'Lemma':<20} {'POS':<10} {'Tag':<10} {'Entity Type':<15}")

print("-" \* 75)

for token in doc:

print(f"{token.text:<20} {token.lemma\_:<20} {token.pos\_:<10} {token.tag\_:<10} {token.ent\_type\_:<15}")

**set 6**

1

import nltk

from nltk.tokenize import word\_tokenize

from nltk import pos\_tag

# Download the necessary NLTK data (run this once)

nltk.download('punkt')

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

# Define the sentences to analyze

sentences = [

"The sun is shining brightly",

"I love reading interesting books"

]

# Function to perform POS tagging

def pos\_tagging(sentence):

# Tokenize the sentence

tokens = word\_tokenize(sentence)

# Perform POS tagging

tagged = pos\_tag(tokens)

return tagged

# Analyze and display POS tags for each sentence

for sentence in sentences:

print(f"Sentence: '{sentence}'")

tagged = pos\_tagging(sentence)

print("POS Tags:")

for word, tag in tagged:

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

print("\n" + "-"\*40 + "\n")

2

import spacy

# Load the spaCy model

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

# Define the sentences to analyze

sentences = [

"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."

]

# Function to perform NER

def perform\_ner(sentence):

# Process the sentence with spaCy

doc = nlp(sentence)

# Extract named entities and their labels

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

return entities

# Analyze and display named entities for each sentence

for sentence in sentences:

print(f"Sentence: '{sentence}'")

entities = perform\_ner(sentence)

print("Named Entities:")

for entity, label in entities:

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

print("\n" + "-"\*40 + "\n")

3

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

# Define the sentences (documents)

documents = [

"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."

]

# Initialize the TF-IDF Vectorizer

vectorizer = TfidfVectorizer()

# Fit and transform the documents

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

# Calculate the cosine similarity matrix

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

# Display the TF-IDF matrix

print("TF-IDF Matrix:")

print(tfidf\_matrix.toarray())

# Display the Cosine Similarity Matrix

print("\nCosine Similarity Matrix:")

print(cosine\_sim)

# Display similarity between each pair of documents

print("\nDocument Similarity:")

for i in range(len(documents)):

for j in range(len(documents)):

if i != j:

print(f"Document {i+1} and Document {j+1}: Similarity = {cosine\_sim[i, j]:.4f}")

4

import re

# Define the input text and regex pattern

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

pattern = r'\bb\w{3}\b'

# Function to find and extract patterns using regex

def extract\_patterns(text, pattern):

# Find all matches in the text

matches = re.findall(pattern, text)

return matches

# Extract and display the patterns

matches = extract\_patterns(text, pattern)

print("Input Text:")

print(text)

print("\nRegex Pattern:")

print(pattern)

print("\nMatches Found:")

print(matches)

**set7**

1

import nltk

from nltk.tokenize import word\_tokenize

from nltk import pos\_tag

# Download the necessary NLTK data (run this once)

nltk.download('punkt')

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

# Define the sentences to analyze

sentences = [

"The red car stopped at the traffic light",

"She quickly ran to catch the bus"

]

# Function to perform POS tagging

def stochastic\_pos\_tagging(sentence):

# Tokenize the sentence

tokens = word\_tokenize(sentence)

# Perform POS tagging using the probabilistic model

tagged = pos\_tag(tokens)

return tagged

# Analyze and display POS tags for each sentence

for sentence in sentences:

print(f"Sentence: '{sentence}'")

tagged = stochastic\_pos\_tagging(sentence)

print("POS Tags:")

for word, tag in tagged:

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

print("\n" + "-"\*40 + "\n")

2

from transformers import pipeline

# Define the text to summarize

text = ("The World Health Organization (WHO) plays a vital role in global health. "

"WHO is headquartered in Geneva, Switzerland, and it is responsible for coordinating "

"international efforts to control and prevent the spread of diseases. Its mission is "

"to promote and protect the health of people worldwide.")

# Initialize the summarization pipeline

summarizer = pipeline("summarization")

# Generate the summary

summary = summarizer(text, max\_length=50, min\_length=25, do\_sample=False)

# Print the original text and the summary

print("Original Text:")

print(text)

print("\nSummary:")

print(summary[0]['summary\_text'])

3

import spacy

from nltk.corpus import wordnet

from nltk.tokenize import word\_tokenize

from nltk.corpus import wordnet as wn

# Load spaCy model

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

# Define sentences

sentences = [

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

"She is an excellent chef and loves to cook delicious meals.",

"The Eiffel Tower in Paris is a famous landmark."

]

# Function to extract noun phrases from a sentence

def extract\_noun\_phrases(sentence):

doc = nlp(sentence)

return [chunk.text for chunk in doc.noun\_chunks]

# Function to get WordNet meanings for a word

def get\_wordnet\_meaning(word):

synsets = wn.synsets(word)

meanings = [synset.definition() for synset in synsets]

return meanings if meanings else ["No meaning found"]

# Process each sentence

for sentence in sentences:

print(f"Sentence: '{sentence}'")

# Extract noun phrases

noun\_phrases = extract\_noun\_phrases(sentence)

print("Noun Phrases:")

for phrase in noun\_phrases:

print(f" {phrase}")

# Get meanings for each noun phrase (first word in the phrase for simplicity)

words = word\_tokenize(phrase)

for word in words:

meanings = get\_wordnet\_meaning(word.lower())

print(f" Word: '{word}'")

print(f" Meanings: {meanings}")

print("\n" + "-"\*40 + "\n")

4

import nltk

from nltk.tokenize import word\_tokenize

from nltk.stem import WordNetLemmatizer

# Download necessary NLTK data (run these lines once if needed)

nltk.download('punkt')

nltk.download('wordnet')

# Initialize the lemmatizer

lemmatizer = WordNetLemmatizer()

# Define sample sentences

sentences = [

"The quick brown foxes jumped over the lazy dogs.",

"I am running in the park with my friends."

]

# Function to perform lemmatization

def lemmatize\_sentence(sentence):

tokens = word\_tokenize(sentence)

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

return lemmatized\_tokens

# Process and display results for each sentence

for sentence in sentences:

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

lemmatized\_tokens = lemmatize\_sentence(sentence)

print("Lemmatized Tokens:")

print(lemmatized\_tokens)

print("\n" + "-"\*40 + "\n")

**set 8**

1

import re

# Define the patterns and their corresponding POS tags

patterns = [

(r'\b(?:The|the)\b', 'DET'), # Matches 'The' or 'the' as determiners

(r'\b(?:cat|dog)\b', 'NOUN'), # Matches 'cat' or 'dog' as nouns

(r'\b(?:is|am|are)\b', 'VERB'), # Matches 'is', 'am', or 'are' as verbs

(r'\b(?:quickly|brightly)\b', 'ADV'), # Matches 'quickly' or 'brightly' as adverbs

(r'\b(?:[A-Za-z]+)\b', 'NOUN') # Matches any other words as nouns

]

# Function to perform POS tagging

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

tokens = text.split()

tagged\_tokens = []

for token in tokens:

tagged = False

for pattern, tag in patterns:

if re.fullmatch(pattern, token):

tagged\_tokens.append((token, tag))

tagged = True

break

if not tagged:

tagged\_tokens.append((token, 'UNKNOWN')) # Tag unknown words as 'UNKNOWN'

return tagged\_tokens

# Define a sample text

text = "The quick brown fox quickly jumped over the lazy dog. The cat is playing with the dog."

# Perform POS tagging

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

# Print results

print("Tagged Tokens:")

for token, tag in tagged\_tokens:

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

2

import spacy

# Load the pre-trained spaCy model for NER

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

# Define the text

text = "The capital of France is Paris, and it's known for the Eiffel Tower."

# Process the text

doc = nlp(text)

# Extract and print named entities

print("Named Entities, Phrases, and Categories:")

for ent in doc.ents:

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

3

import nltk

from nltk import PCFG

from nltk.parse import ViterbiParser

# Define the PCFG grammar

pcfg\_grammar = PCFG.fromstring("""

S -> NP VP [1.0]

NP -> Det Adj N [0.4] | Det Adj Adj N [0.1] | NP PP [0.4] | Det N [0.5]

VP -> V NP [0.7] | VP PP [0.3]

PP -> P NP [1.0]

Det -> 'the' [0.7] | 'a' [0.3]

N -> 'fox' [0.4] | 'dog' [0.3] | 'cat' [0.2] | 'bird' [0.1]

V -> 'jumps' [0.5] | 'runs' [0.3] | 'sits' [0.2]

P -> 'over' [0.6] | 'on' [0.4]

Adj -> 'quick' [0.5] | 'lazy' [0.3] | 'brown' [0.2]

""")

# Create a parser using the PCFG grammar

parser = ViterbiParser(pcfg\_grammar)

# Define the input string

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

# Tokenize the sentence

tokens = nltk.word\_tokenize(sentence.lower()) # Lowercase to match grammar

# Parse the tokens

print("Possible parses:")

for tree in parser.parse(tokens):

print(tree)

tree.pretty\_print()

4

import nltk

from nltk.stem import PorterStemmer

from nltk.tokenize import word\_tokenize

# Ensure you have the necessary NLTK resources

nltk.download('punkt')

# Initialize the Porter Stemmer

stemmer = PorterStemmer()

# Sample sentences

sentences = [

"Coding with Python is very enjoyable.",

"I had a delicious meal at the restaurant."

]

# Function to perform stemming on a list of words

def stem\_words(words):

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

# Process each sentence

for sentence in sentences:

# Tokenize the sentence into words

words = word\_tokenize(sentence.lower())

# Perform stemming

stemmed\_words = stem\_words(words)

# Print the results

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

print(f"Stemmed Words: {stemmed\_words}")

print()

**set 9**

1

import nltk

from nltk.stem import PorterStemmer

from nltk.tokenize import word\_tokenize

# Ensure you have the necessary NLTK resources

nltk.download('punkt')

# Initialize the Porter Stemmer

stemmer = PorterStemmer()

# Sample sentences

sentences = [

"Coding with Python is very enjoyable.",

"I had a delicious meal at the restaurant."

]

# Function to perform stemming on a list of words

def stem\_words(words):

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

# Process each sentence

for sentence in sentences:

# Tokenize the sentence into words

words = word\_tokenize(sentence.lower())

# Perform stemming

stemmed\_words = stem\_words(words)

# Print the results

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

print(f"Stemmed Words: {stemmed\_words}")

print()

2

from textblob import TextBlob

# Sample text data

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

def analyze\_sentiment(text):

# Create a TextBlob object

blob = TextBlob(text)

# Get the sentiment polarity

polarity = blob.sentiment.polarity

# Determine sentiment based on polarity

if polarity > 0:

sentiment = 'Positive'

elif polarity < 0:

sentiment = 'Negative'

else:

sentiment = 'Neutral'

return sentiment

# Analyze the sentiment of the sample text

sentiment = analyze\_sentiment(text)

# Print the result

print(f"Text: {text}")

print(f"Sentiment: {sentiment}")

3

import re

def recognize\_dialog\_act(sentence):

# Define patterns for different dialog acts

patterns = {

'Greeting': r'\bhello\b|\bhi\b|\bhey\b|\bgood morning\b|\bgood afternoon\b|\bgood evening\b',

'Response': r'\bI am\b|\bI am doing\b|\bthank you\b|\byou are welcome\b|\bI\'m doing\b',

'Request': r'\bplease\b.\*\bpass\b|\bcan you\b.\*\bpass\b|\bmay I\b.\*\bplease\b',

'Inquiry': r'\bhow are you\b|\bwhat is\b|\bwhat time\b|\bwhen\b|\bwhere\b|\bwho\b|\bcan\b.\*\bplease\b',

'Statement': r'\bthe meeting is\b|\bthis is\b|\bthere is\b'

}

# Lowercase the sentence for case-insensitive matching

sentence\_lower = sentence.lower()

# Check each pattern and return the corresponding dialog act

for act, pattern in patterns.items():

if re.search(pattern, sentence\_lower):

return act

return 'Unknown'

# Define the dialog

dialog = [

"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."

]

# Recognize and print dialog acts for each sentence

for sentence in dialog:

act = recognize\_dialog\_act(sentence)

print(f"Sentence: '{sentence}'")

print(f"Dialog Act: {act}")

print()

4

import re

class DateFSA:

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

# Regular expression for DD/MM/YYYY format

self.date\_pattern = re.compile(r"^(0[1-9]|[12][0-9]|3[01])/(0[1-9]|1[0-2])/\d{4}$")

def is\_valid\_date(self, date\_string):

"""

Check if the given date string matches the DD/MM/YYYY format.

"""

if self.date\_pattern.match(date\_string):

return True

return False

# Create an instance of DateFSA

date\_fsa = DateFSA()

# Sample dates to test

dates = [

"01/01/2024", # Valid date

"31/12/2024", # Valid date

"30/02/2024", # Invalid date (February 30th doesn't exist)

"15/13/2024", # Invalid date (Month 13 doesn't exist)

"2024/12/31", # Invalid format

"12/25/2024" # Invalid format (Incorrect month/day positions)

]

# Test the dates

for date in dates:

is\_valid = date\_fsa.is\_valid\_date(date)

print(f"Date: {date} - Valid: {is\_valid}")

from datetime import datetime

class DateFSA:

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

self.date\_pattern = re.compile(r"^(0[1-9]|[12][0-9]|3[01])/(0[1-9]|1[0-2])/\d{4}$")

def is\_valid\_date(self, date\_string):

if self.date\_pattern.match(date\_string):

day, month, year = map(int, date\_string.split('/'))

try:

datetime(year, month, day)

return True

except ValueError:

return False

return False

# Create an instance of DateFSA

date\_fsa = DateFSA()

# Sample dates to test

dates = [

"01/01/2024", # Valid date

"31/12/2024", # Valid date

"30/02/2024", # Invalid date (February 30th doesn't exist)

"15/13/2024", # Invalid date (Month 13 doesn't exist)

"2024/12/31", # Invalid format

"12/25/2024" # Invalid format (Incorrect month/day positions)

]

# Test the dates

for date in dates:

is\_valid = date\_fsa.is\_valid\_date(date)

print(f"Date: {date} - Valid: {is\_valid}")from sklearn.feature\_extraction.text import TfidfVectorizer

**set-10**

1

import numpy as np

# Sample documents

documents = [

"Natural language processing (NLP) is a field of study in artificial intelligence.",

"NLP techniques are used in various applications like machine translation and sentiment analysis.",

"The development of NLP tools and libraries has made text analysis easier."

]

# Create the TF-IDF vectorizer

vectorizer = TfidfVectorizer()

# Fit and transform the documents

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

# Define a sample query

query = "NLP applications in text analysis"

# Transform the query using the same vectorizer

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

# Compute the cosine similarity between the query and each document

from sklearn.metrics.pairwise import cosine\_similarity

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

# Display the similarity scores

print("Cosine Similarity Scores:")

for i, score in enumerate(similarity\_scores):

print(f"Document {i + 1}: {score:.4f}")

# Find the most relevant document

most\_relevant\_index = np.argmax(similarity\_scores)

print(f"\nMost relevant document is Document {most\_relevant\_index + 1}")

2

import spacy

# Load the spaCy model

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

# Sample sentences

sentences = [

"John and Mary went to the store.",

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

]

def parse\_dependencies(sentence):

# Parse the sentence using spaCy

doc = nlp(sentence)

# Extract and print dependency relations

for token in doc:

print(f"Token: {token.text}")

print(f" Lemma: {token.lemma\_}")

print(f" POS: {token.pos\_}")

print(f" Dependency: {token.dep\_}")

print(f" Head: {token.head.text}")

print()

# Parse each sentence

for sentence in sentences:

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

parse\_dependencies(sentence)

print("="\*50)

3

from transformers import MarianMTModel, MarianTokenizer

# Load the model and tokenizer

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

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

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

def translate\_text(text):

# Tokenize the input text

inputs = tokenizer(text, return\_tensors="pt", padding=True, truncation=True)

# Perform translation

with torch.no\_grad():

translated = model.generate(\*\*inputs)

# Decode the translated text

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

return translated\_text

# Sample text to translate

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

# Perform translation

german\_translation = translate\_text(english\_text)

# Print the result

print(f"English Text: {english\_text}")

print(f"German Translation: {german\_translation}")

4

# Define the variables and their truth values

variables = {

'p': True,

'q': True,

'r': False

}

# Define the expressions to evaluate

expressions = [

"p and q",

"p or r",

"not p",

"q and (r or p)"

]

def evaluate\_expression(expression, variables):

"""

Evaluates a logical expression based on the given variables.

"""

# Create a local namespace with the variable values

local\_namespace = variables.copy()

# Replace logical operators with Python equivalents

expression = expression.replace("and", " and ").replace("or", " or ").replace("not", " not ")

try:

# Evaluate the expression

result = eval(expression, {}, local\_namespace)

except Exception as e:

result = f"Error evaluating expression: {e}"

return result

# Evaluate each expression and print the result

for expr in expressions:

result = evaluate\_expression(expr, variables)

print(f"Expression: {expr}")

print(f"Result: {result}")

print("-" \* 30)