**1. Write a program to demonstrate how to use regular expressions in Python to match and**

**search for patterns in text.**

import re

text = "The quick brown fox jumps over the lazy dog. The fox is clever."

pattern = r"fox"

match = re.search(pattern, text)

if match:

print(f"'fox' found at position: {match.start()}")

matches = re.findall(pattern, text)

print(f"Occurrences of 'fox': {matches}")

if re.match(r"The", text):

print("The text starts with 'The'.")

new\_text = re.sub(pattern, "cat", text)

print(f"Modified text: {new\_text}")

**2.Implement a basic finite state automaton that recognizes a specific language or pattern. In**

**this example, we&#39;ll create a simple automaton to match strings ending with &#39;ab&#39; using python.**

def fsa\_match\_ab(string):

state = 'q0'

for char in string:

if state == 'q0':

if char == 'a':

state = 'q1'

elif state == 'q1':

if char == 'b':

state = 'q2'

elif char != 'a':

state = 'q0'

elif state == 'q2':

if char == 'a':

state = 'q1'

else:

state = 'q0'

return "Accepted" if state == 'q2' else "Rejected"

test\_strings = ["ab", "aab", "baba", "abc", "abab", "ba", "b"]

for s in test\_strings:

print(f"String '{s}' → {fsa\_match\_ab(s)}")

**3. Write program demonstrates how to perform morphological analysis using the NLTK**

**library in Python**.

import re

def simple\_tokenize(text):

return re.findall(r'\b\w+\b', text)

def simple\_stem(word):

suffixes = ('ing', 'ed', 'es', 's')

for suffix in suffixes:

if word.endswith(suffix):

return word[:-len(suffix)]

return word

text = "Running wolves are faster than jumping foxes."

words = simple\_tokenize(text)

print("Original Words:", words)

stemmed\_words = [simple\_stem(word) for word in words]

print("Stemmed Words:", stemmed\_words)

**4. Implement a finite-state machine for morphological parsing. In this example, we&#39;ll reate**

**a simple machine to generate plural forms of English nouns using python.**

class FiniteStateMachine:

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

self.transitions = {

"singular": {

"s": "plural",

"x": "plural",

"z": "plural",

"ch": "plural",

"sh": "plural",

"y": "replace\_y\_plural",

"default": "plural\_s"

},

"replace\_y\_plural": {

"default": "plural\_ies"

}

}

def process(self, word):

for ending in ["s", "x", "z", "ch", "sh"]:

if word.endswith(ending):

return word + "es"

if word.endswith("y") and word[-2] not in "aeiou":

return word[:-1] + "ies"

return word + "s"

fsm = FiniteStateMachine()

nouns = ["cat", "bus", "fox", "baby", "dog"]

plural\_forms = {noun: fsm.process(noun) for noun in nouns}

print(plural\_forms)

**5. Use the Porter Stemmer algorithm to perform word stemming on a list of words using**

**python libraries.**

import nltk

from nltk.stem import PorterStemmer

stemmer=PorterStemmer()

words=["running","studying","watchinmgh"]

stemmed\_words={word:stemmer.stem(word) for word in words}

print(stemmed\_words)

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

**bigram model using python.**

import random

from collections import defaultdict

def train\_bigram\_model(text):

words = text.split()

bigrams = defaultdict(list)

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

bigrams[words[i]].append(words[i + 1])

return bigrams

def generate\_text(bigrams, start\_word, length=10):

word = start\_word

result = [word]

for \_ in range(length - 1):

if word in bigrams:

word = random.choice(bigrams[word])

result.append(word)

else:

break

return ' '.join(result)

# Example usage

text\_corpus = "this is a simple example of a bigram model for text generation"

bigram\_model = train\_bigram\_model(text\_corpus)

print(generate\_text(bigram\_model, "this", 10))

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

import random

from collections import defaultdict

training\_data = [

("Natural", "JJ"), ("Language", "NN"), ("Processing", "NN"),

("is", "VBZ"), ("a", "DT"), ("fascinating", "JJ"),

("field", "NN"), ("of", "IN"), ("Artificial", "JJ"),

("Intelligence", "NN"), (".", ".")

]

def train\_pos\_tagger(training\_data):

tag\_counts = defaultdict(lambda: defaultdict(int))

for word, tag in training\_data:

tag\_counts[word][tag] += 1

pos\_model = {word: max(tags, key=tags.get) for word, tags in tag\_counts.items()}

return pos\_model

def stochastic\_pos\_tagger(text, pos\_model, default\_tag="NN"):

words = text.split()

tagged\_words = []

for word in words:

tag = pos\_model.get(word, random.choice(["NN", "VB", "JJ", "RB", "IN", "DT", "VBZ"]))

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

return tagged\_words

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

pos\_model = train\_pos\_tagger(training\_data)

test\_sentence = "Artificial Intelligence is amazing"

tagged\_output = stochastic\_pos\_tagger(test\_sentence, pos\_model)

print("POS Tagging Result:")

for word, tag in tagged\_output:

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

**8. Implement a simple stochastic part-of-speech tagging algorithm using a basic**

**probabilistic model to assign POS tags using python.**

import random

from collections import defaultdict

training\_data = [

("Natural", "JJ"), ("Language", "NN"), ("Processing", "NN"),

("is", "VBZ"), ("a", "DT"), ("fascinating", "JJ"),

("field", "NN"), ("of", "IN"), ("Artificial", "JJ"),

("Intelligence", "NN"), (".", ".")

]

def train\_pos\_tagger(training\_data):

tag\_counts = defaultdict(lambda: defaultdict(int))

for word, tag in training\_data:

tag\_counts[word][tag] += 1

pos\_model = {word: max(tags, key=tags.get) for word, tags in tag\_counts.items()}

return pos\_model

def stochastic\_pos\_tagger(text, pos\_model, default\_tag="NN"):

words = text.split()

tagged\_words = []

for word in words:

tag = pos\_model.get(word, random.choice(["NN", "VB", "JJ", "RB", "IN", "DT", "VBZ"]))

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

return tagged\_words

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

pos\_model = train\_pos\_tagger(training\_data)

test\_sentence = "Artificial Intelligence is amazing"

tagged\_output = stochastic\_pos\_tagger(test\_sentence, pos\_model)

print("POS Tagging Result:")

for word, tag in tagged\_output:

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

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

**python.**

import re

patterns = [

(r'\b[Ii]s\b', 'VERB'),

(r'\b[Aa]re\b', 'VERB'),

(r'\b[Tt]he\b', 'DET'),

(r'\b[Aa]n?\b', 'DET'),

(r'\b[cC]at\b', 'NOUN'),

(r'\b[dD]og\b', 'NOUN'),

(r'\b[jJ]umps?\b', 'VERB')

]

def pos\_tag(sentence):

words = sentence.split()

tagged\_sentence = []

for word in words:

tag = 'UNKNOWN'

for pattern, pos in patterns:

if re.fullmatch(pattern, word):

tag = pos

break

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

return tagged\_sentence

# Example usage

sentence = "The cat jumps over the dog"

print(pos\_tag(sentence))

**10. Implement transformation-based tagging using a set of transformation rules, apply a**

**simple rule to tag words using python.**

import re

def initial\_pos\_tagging(words):

return [(word, "NN") for word in words]

def apply\_transformation\_rules(tagged\_words):

transformed\_tags = []

for word, tag in tagged\_words:

if word.lower() in ["is", "am", "are"]:

transformed\_tags.append((word, "VB"))

elif re.match(r".\*ing$", word):

transformed\_tags.append((word, "VBG"))

elif word.lower() in ["a", "an", "the"]:

transformed\_tags.append((word, "DT"))

elif re.match(r".\*ly$", word):

transformed\_tags.append((word, "RB"))

elif word[0].isupper():

transformed\_tags.append((word, "NNP"))

else:

transformed\_tags.append((word, tag))

return transformed\_tags

def transformation\_based\_tagger(text):

words = text.split()

tagged\_words = initial\_pos\_tagging(words)

transformed\_tags = apply\_transformation\_rules(tagged\_words)

return transformed\_tags

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

sentence = "The cat is running quickly in the garden"

tagged\_output = transformation\_based\_tagger(sentence)

print("POS Tagging Result:")

for word, tag in tagged\_output:

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

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

class TopDownParser:

def \_\_init\_\_(self, grammar, start\_symbol):

self.grammar = grammar

self.start\_symbol = start\_symbol

def parse(self, tokens, symbol=None, pos=0):

if symbol is None:

symbol = self.start\_symbol

if pos >= len(tokens):

return pos if symbol == "" else -1

if symbol in self.grammar: # Non-terminal

for production in self.grammar[symbol]:

new\_pos = pos

success = True

for sym in production:

new\_pos = self.parse(tokens, sym, new\_pos)

if new\_pos == -1:

success = False

break

if success:

return new\_pos

return -1 # No valid production found

else: # Terminal

return pos + 1 if pos < len(tokens) and tokens[pos] == symbol else -1

def main():

grammar = {

"S": [["NP", "VP"]],

"NP": [["Det", "Noun"]],

"VP": [["Verb", "NP"], ["Verb"]],

"Det": [["the"], ["a"]],

"Noun": [["cat"], ["dog"]],

"Verb": [["chased"], ["saw"]]

}

parser = TopDownParser(grammar, "S")

sentence = ["the", "cat", "chased", "a", "dog"]

result = parser.parse(sentence)

if result == len(sentence):

print("Valid sentence")

else:

print("Invalid sentence")

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

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

**class EarleyParser:**

**def \_\_init\_\_(self, grammar):**

**self.grammar = grammar # The CFG in dictionary form**

**def parse(self, sentence):**

**words = sentence.split()**

**n = len(words)**

**chart = [set() for \_ in range(n + 1)] # Parsing table (chart)**

**# Convert grammar to list of production rules**

**grammar\_rules = [(lhs, rhs.split()) for lhs, rhs in self.grammar.items()]**

**# Initialize chart with start rule**

**for lhs, rhs in grammar\_rules:**

**if lhs == "S": # Start symbol**

**chart[0].add((lhs, tuple(rhs), 0, 0))**

**# Parsing process**

**for i in range(n + 1):**

**changes = True**

**while changes: # Keep processing until no changes**

**changes = False**

**for state in list(chart[i]):**

**lhs, rhs, dot, start = state**

**if dot < len(rhs): # Prediction and Scanning**

**symbol = rhs[dot]**

**if symbol in self.grammar: # Non-terminal (Prediction)**

**for rule\_lhs, rule\_rhs in grammar\_rules:**

**if rule\_lhs == symbol:**

**new\_state = (rule\_lhs, tuple(rule\_rhs), 0, i)**

**if new\_state not in chart[i]:**

**chart[i].add(new\_state)**

**changes = True**

**elif i < n and words[i] == symbol: # Terminal (Scanning)**

**new\_state = (lhs, rhs, dot + 1, start)**

**if new\_state not in chart[i + 1]:**

**chart[i + 1].add(new\_state)**

**changes = True**

**else: # Completion**

**for state\_ in list(chart[start]):**

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

**program.**

import nltk

from nltk import CFG

# Define the grammar

grammar = CFG.fromstring("""

S -> NP VP

NP -> Det N

VP -> V NP

Det -> 'the'

N -> 'cat' | 'dog'

V -> 'chased' | 'saw'

""")

# Define the sentence

sentence = "the cat chased the dog".split()

# Create a parser

parser = nltk.ChartParser(grammar)

# Parse and display the tree

for tree in parser.parse(sentence):

tree.pretty\_print()

**14.Create a program in python to check for agreement in sentences based on a context-free**

**grammar&#39;s rules.**

import nltk

from nltk import CFG

# Define a Context-Free Grammar with agreement rules

grammar = CFG.fromstring("""

S -> NP VP

NP -> Det N\_sing | Det N\_plur

VP -> V\_sing | V\_plur

Det -> 'the'

N\_sing -> 'cat' | 'dog'

N\_plur -> 'cats' | 'dogs'

V\_sing -> 'runs' | 'barks'

V\_plur -> 'run' | 'bark'

""")

# Function to check if a sentence follows subject-verb agreement

def check\_agreement(sentence):

words = sentence.split() # Tokenize sentence

parser = nltk.ChartParser(grammar) # Create parser

# Check if the sentence is valid in the given grammar

try:

parse\_trees = list(parser.parse(words))

if parse\_trees:

print(f"✅ The sentence '{sentence}' is grammatically correct!")

for tree in parse\_trees:

tree.pretty\_print()

else:

print(f"❌ The sentence '{sentence}' is incorrect.")

except ValueError:

print(f"❌ The sentence '{sentence}' is incorrect.")

# Test cases

check\_agreement("the cat runs") # ✅ Valid

check\_agreement("the cats run") # ✅ Valid

check\_agreement("the cat run") # ❌ Invalid (Singular subject, plural verb)

check\_agreement("the cats runs") # ❌ Invalid (Plural subject, singular verb)

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

import nltk

from nltk import PCFG

# Define a Probabilistic Context-Free Grammar (PCFG)

pcfg\_grammar = PCFG.fromstring("""

S -> NP VP [0.95] | VP [0.05]

NP -> Det N [0.6] | N [0.4]

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

Det -> 'the' [1.0]

N -> 'dog' [0.7] | 'cat' [0.3]

V -> 'barked' [0.8] | 'chased' [0.2]

""")

# Create a Viterbi parser

parser = nltk.ViterbiParser(pcfg\_grammar)

# Define the sentence to parse

sentence = "the dog barked".split()

# Parse the sentence and display the most probable parse tree

for tree in parser.parse(sentence):

tree.pretty\_print()

**16. Implement a Python program using the SpaCy library to perform Named Entity**

**Recognition (NER) on a given text.**

import spacy

# Load SpaCy's pre-trained English model

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

# Input text for NER

text = "Apple was founded by Steve Jobs in Cupertino in 1976."

# Process the text using SpaCy

doc = nlp(text)

# Extract and print named entities

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

for ent in doc.ents:

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

**17. Write program demonstrates how to access WordNet, a lexical database, to retrieve**

**synsets and explore word meanings in python.**

import nltk

from nltk.corpus import wordnet as wn

# Download WordNet data (if not already installed)

nltk.download('wordnet')

# Input word

word = "bank"

# Retrieve synsets for the word

synsets = wn.synsets(word)

# Display synsets and their meanings

print(f"Synsets for the word '{word}':")

for synset in synsets:

print(f"\nSynset: {synset.name()}")

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

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

**20.** **Implement a basic information retrieval system using TF-IDF (Term Frequency-Inverse**

**Document Frequency) for document ranking using python.**

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine\_similarity

import nltk

# Download NLTK stopwords

nltk.download('stopwords')

from nltk.corpus import stopwords

# Sample Documents

documents = [

"The cat in the hat.",

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

"The dog barks at the cat in the park.",

"Foxes are wild animals that live in forests."

]

# Query

query = "cat in the park"

# Step 1: Preprocess the documents and query by removing stopwords

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

# Function to remove stopwords

def preprocess(text):

return ' '.join([word for word in text.lower().split() if word not in stop\_words])

# Preprocess documents and query

documents = [preprocess(doc) for doc in documents]

query = preprocess(query)

# Step 2: Create the TF-IDF model

vectorizer = TfidfVectorizer()

# Fit the model on the documents

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

# Step 3: Convert the query to the same vector space

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

# Step 4: Calculate cosine similarities between

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

import spacy

import neuralcoref

# Load the spaCy language model

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

# Add neuralcoref to the pipeline for coreference resolution

neuralcoref.add\_to\_pipe(nlp)

# Function to perform reference resolution in a text

def resolve\_references(text):

# Process the text with spaCy NLP pipeline

doc = nlp(text)

# Resolve coreferences

resolved\_text = doc.\_.coref\_resolved

return resolved\_text

# Test the program

text = "John went to the store. He bought some milk."

resolved\_text = resolve\_references(text)

print("Original Text: ", text)

print("Resolved Text: ", resolved\_text)

**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.**

import openai

# Set your OpenAI API key

openai.api\_key = "your-api-key-here" # Replace with your actual API key

# Function to generate text using GPT-3

def generate\_text(prompt):

response = openai.Completion.create(

engine="text-davinci-003", # You can use different engines like "text-curie-001" or "text-ada-001"

prompt=prompt,

max\_tokens=100, # The maximum number of tokens (words) you want in the output

temperature=0.7, # Controls randomness: 0 is deterministic, 1 is very random

n=1, # The number of responses to generate

stop=None # Optionally define a stopping sequence

)

# Get the generated text

generated\_text = response.choices[0].text.strip()

return generated\_text

# Test the function with a prompt

prompt = "Once upon a time in a distant kingdom, there was a brave knight who"

generated\_text = generate\_text(prompt)

print("Generated Text:")

print(generated\_text)