1 Question:

Given a context-free grammar (CFG) in Chomsky Normal Form (CNF):

S -> AB | BC

A -> BA | a

B ->CC | b

C-> AB | a

and a string \*\*"baaba"\*\*, use the \*\*CYK algorithm\*\* to check if the string can be generated by this grammar.

Answer

1. The string **"baaba"** has length n=5n = 5n=5, so we need a **5x5 table**

**Step 2: Populate the Table for Length 1 Substrings (Terminals)**

Fill in the table with the variables that produce each single terminal symbol.

* For "b" at position 1: BBB
* For "a" at position 2: A, CA,
* For "a" at position 3: A, CA,
* For "b" at position 4: BBB
* For "a" at position 5: A, CA,

|  | **1** | **2** | **3** | **4** | **5** |
| --- | --- | --- | --- | --- | --- |
| **1** | B | A, C | A, C | B | A, C |

**Step 3: Fill in the Table for Longer Substrings**

We'll build up substrings of increasing lengths, checking for possible variable combinations that derive each substring.

**Length 2**

1. For positions (1, 2): Substring "ba"
   * Possible pairs: (B,A),(B,C)(B, A), (B, C)(B,A),(B,C)
   * Using grammar rules:
     + B→BA→AB -> BA -> AB→BA→A
   * Result: A, C

2 Question

Consider a weather forecasting system modelled as a Hidden Markov Model. The weather can be either Sunny (S) or Rainy (R), but we can only observe the activities: Walking (W), Shopping (Sh), or Cleaning (C). The HMM parameters are:

* Hidden States (Weather): Sunny (S), Rainy (R)
* Observed States (Activities): Walking (W), Shopping (Sh), Cleaning (C)

The parameters of the model are:

* Initial state probabilities:
  + P(S)=0.6P(S) = 0.6P(S)=0.6, P(R)=0.4P(R) = 0.4P(R)=0.4
* State transition probabilities:
  + P(S→S) =0.7 P (S ->S) = 0.7P(S→S) =0.7, P(S→R) =0.3P(S ->R) = 0.3P(S→R)=0.3
  + P(R→R) =0.6 P (R ->R) = 0.6P(R→R) =0.6, P(R→S) =0.4P(R ->S) = 0.4P(R→S)=0.4
* Observation probabilities:
  + When Sunny: P(W∣S) =0.6P(W|S) = 0.6P(W∣S) =0.6, P(Sh∣S) =0.3P(Sh|S) = 0.3P(Sh∣S) =0.3, P(C∣S) =0.1P(C|S) = 0.1P(C∣S) =0.1
  + When Rainy: P(W∣R) =0.1P(W|R) = 0.1P(W∣R) =0.1, P(Sh∣R) =0.4P(Sh|R) = 0.4P(Sh∣R) =0.4, P(C∣R) =0.5P(C|R) = 0.5P(C∣R) =0.5

The following sequence of observed activities: **"Walking, Shopping, Cleaning"**. a sequence of **observed activities** (e.g., Walking, Shopping, Cleaning) and determine which sequence of **hidden weather conditions** (e.g., Sunny, Rainy) is most likely sequence of the hidden process (weather) that explains the observed events.

**Answer 2**

**Enumerate Possible Weather Sequences**

Since there are three activities (Walking, Shopping, Cleaning), there are 23=8 possible sequences of weather states (Sunny or Rainy) that could explain the observations.

The sequences are:

1. Sunny, Sunny, Sunny
2. Sunny, Sunny, Rainy
3. Sunny, Rainy, Sunny
4. Sunny, Rainy, Rainy
5. Rainy, Sunny, Sunny
6. Rainy, Sunny, Rainy
7. Rainy, Rainy, Sunny
8. Rainy, Rainy, Rainy

**2. Calculate the Probability of Each Sequence**

For each sequence, we’ll calculate the probability by combining:

* The initial state probability.
* The transition probabilities between states.
* The observation probabilities given the state.

**Example Calculations**

Let’s calculate the probability for the first few sequences. We’ll denote:

* **P(S)** = Initial probability of Sunny
* **P(R)** = Initial probability of Rainy
* **P(S→S) P(S -> S)** = Transition probabilities from Sunny to Sunny and Sunny to Rainy, respectively.
* **P(W∣S)** **P(Sh∣S)** and **P(C∣S)** = Observation probabilities when Sunny

Calculate for the first two sequences as examples.

**Sequence 1: Sunny, Sunny, Sunny**

1. Initial state P(S)=0.6
2. Observation Walking with Sunny: P(W∣S)=0.6
3. Transition from Sunny to Sunny: P(S→S)=0.7
4. Observation Shopping with Sunny: P(Sh∣S)=0.3
5. Transition from Sunny to Sunny: P(S→S)=0.7
6. Observation Cleaning with Sunny: P(C∣S)=0.1

So, the probability for **Sunny, Sunny, Sunny** is:

0.6×0.6×0.7×0.3×0.7×0.1=0.0052920.6

**Sequence 2: Sunny, Sunny, Rainy**

1. Initial state P(S)=0.6
2. Observation Walking with Sunny: P(W∣S) =0.6
3. Transition from Sunny to Sunny: P(S→S) =0.7P
4. Observation Shopping with Sunny: P(Sh∣S) =0.3
5. Transition from Sunny to Rainy: P(S→R) =0.3
6. Observation Cleaning with Rainy: P(C∣R) =0.5

So, the probability for **Sunny, Sunny, Rainy** is:

0.6×0.6×0.7×0.3×0.3×0.5=0.0189

**3. Repeat for All Sequences**

Following this process, calculate the probability for each of the 8 possible sequences.

**4. Identify the Sequence with the Highest Probability**

Once all probabilities are calculated, choose the sequence with the highest probability as the most likely sequence of weather states.

Based on our previous Viterbi calculation, **Sunny, Sunny, Rainy** was identified as the most probable sequence, with a probability of 0.0189. This method confirms that **Sunny, Sunny, Rainy** is indeed the most likely sequence.

Only Information

The **Viterbi algorithm** is a dynamic programming algorithm used to find the most likely sequence of hidden states (in this case, weather conditions like Sunny and Rainy) given a sequence of observed events (like activities: Walking, Shopping, Cleaning) in a Hidden Markov Model (HMM). The algorithm calculates the probability of each possible sequence of states, tracking the path that has the highest probability of leading to the observed sequence.

**Key Concepts in the Viterbi Algorithm**

1. **Hidden States (Weather Conditions)**: These are the actual states we are trying to infer. In our example, these states are **Sunny (S)** and **Rainy (R)**.
2. **Observed States (Activities)**: These are what we can observe directly. In our example, these are **Walking (W)**, **Shopping (Sh)**, and **Cleaning (C)**.
3. **Transition Probabilities**: These are the probabilities of moving from one hidden state to another, such as the probability of going from Sunny to Rainy.
4. **Observation Probabilities**: These are the probabilities of observing a particular activity given the current hidden state, like the probability of walking when it’s Sunny.

**Steps of the Viterbi Algorithm**

The Viterbi algorithm involves three main steps: **Initialization**, **Recursion**, and **Termination/Backtracking**.

**1. Initialization**

For the first observation, calculate the probability of starting in each hidden state and observing the given activity. This is done by multiplying the **initial probability** of each state by the **observation probability** for the first observed activity.

For example, if our first observed activity is Walking:

* V(S,1) =P(S)×P(W∣S)
* V(R,1) =P(R)×P(W∣R)

**2. Recursion**

For each subsequent observation, calculate the probability of each hidden state by considering all possible transitions from the previous states, and multiply by the observation probability of the current activity. At each step, we keep track of the **highest probability** path leading to each state.

For example, if the second observed activity is Shopping, the calculation to find the most likely path to each state (Sunny or Rainy) would be:

* V(S,2)=max (V(S,1)×P(S→S),V(R,1)×P(R→S))×P(Sh∣S)
* V(R,2)=max (V(S,1)×P(S→R),V(R,1)×P(R→R))×P(Sh∣R)
* This recursion step is repeated for each subsequent observation.

**3. Termination and Backtracking**

After processing all observations, we look at the last time step and choose the state with the highest probability. This gives us the end of the most likely sequence of states.

Then, by backtracking through the stored paths, we trace back to determine the most likely sequence of states (the most probable path).

**Why Use the Viterbi Algorithm?**

The Viterbi algorithm efficiently finds the **most probable path** through a sequence of hidden states that explains the observations, rather than calculating the probabilities of all possible paths. It’s especially useful in applications where it’s essential to infer hidden states, such as:

* Weather prediction based on observed activities
* Speech recognition
* Part-of-speech tagging in natural language processing
* DNA sequence alignment in bioinformatics

By keeping only the maximum probability at each step, the Viterbi algorithm avoids the computational explosion that would result from enumerating all possible state sequences.

Question 3

Corpus I'm going to the market at 5 p.m. I'll buy 3 apples, 2 bananas, and 10 oranges. Can't wait to try them! Also, I owe you $50. Let's meet tomorrow at 10:00 a.m.

Write a Python program that performs text normalization on a sentence. The program should:

1. Remove unwanted punctuation (e.g., periods, commas, exclamation marks).
2. Convert the sentence to either lower case or upper case for the entire document.
3. Expand abbreviations (e.g., "can't" → "cannot", "won't" → "will not").
4. Convert numbers into words (e.g., "123" → "one hundred twenty-three").
5. Canonicalization: Apply text canonicalization by ensuring consistency in word usage (e.g., "colour" → "color").

Answer 3

import re

from num2words import num2words

# Dictionary for abbreviation expansion

ABBREVIATIONS = {

"can't": "cannot",

"won't": "will not",

"i'll": "i will",

"i've": "i have",

"i'd": "i would",

"you're": "you are",

"it's": "it is",

"we're": "we are",

"they're": "they are",

"let's": "let us",

"n't": " not",

"'re": " are",

"'s": " is",

"'d": " would",

"'ll": " will",

"'ve": " have"

}

# Dictionary for canonicalization

CANONICALIZATION = {

"colour": "color",

"favour": "favor",

"neighbour": "neighbor",

"organise": "organize",

"realise": "realize",

"analyse": "analyze"

}

# Function to expand abbreviations

def expand\_abbreviations(text):

words = text.split()

expanded\_words = []

for word in words:

word\_lower = word.lower()

if word\_lower in ABBREVIATIONS:

expanded\_words.append(ABBREVIATIONS[word\_lower])

else:

expanded\_words.append(word)

return ' '.join(expanded\_words)

# Function to convert numbers to words

def convert\_numbers\_to\_words(text):

words = text.split()

converted\_words = []

for word in words:

if word.isdigit(): # Check if the word is a digit

converted\_words.append(num2words(int(word)))

else:

converted\_words.append(word)

return ' '.join(converted\_words)

# Function for canonicalization

def canonicalize\_text(text):

words = text.split()

canonicalized\_words = []

for word in words:

word\_lower = word.lower()

if word\_lower in CANONICALIZATION:

canonicalized\_words.append(CANONICALIZATION[word\_lower])

else:

canonicalized\_words.append(word)

return ' '.join(canonicalized\_words)

# Main normalization function

def normalize\_text(text):

# 1. Remove unwanted punctuation

text = re.sub(r'[.,!]', '', text)

# 2. Convert to lower case

text = text.lower()

# 3. Expand abbreviations

text = expand\_abbreviations(text)

# 4. Convert numbers to words

text = convert\_numbers\_to\_words(text)

# 5. Canonicalize text

text = canonicalize\_text(text)

return text

# Example usage

corpus = "I'm going to the market at 5 p.m. I'll buy 3 apples, 2 bananas, and 10 oranges. Can't wait to try them! Also, I owe you $50. Let's meet tomorrow at 10:00 a.m."

normalized\_text = normalize\_text(corpus)

print("Normalized Text:", normalized\_text)

Normalized Text: I m going to the market at five pm i will buy three apples two bananas and ten oranges cannot wait to try them also i owe you $50 let us meet tomorrow at 10:00 am

Question 4

Top-Down Parsing (Recursive Descent Parsing)

Practical Question:

You are given the following context-free grammar (CFG) for simple arithmetic expressions:

* E→T ∣ E+TE ->T | E + TE→T ∣ E+T
* T→F ∣ T∗FT ->F | T \* FT→F ∣ T∗F
* F→(E) ∣ idF ->(E) | idF→(E) ∣ id

Where:

* E represents an expression.
* T represents a term.
* F represents a factor.
* id represents an identifier (e.g., variables or numbers).

Task: Implement a recursive descent parser for this grammar and check whether the following input strings are valid expressions:

1. id + id \* id
2. id \* (id + id)

Top-Down Parsing (Recursive Descent Parsing)

Practical Question:

You are given the same grammar for arithmetic expressions:

* E→T ∣ E+TE \rightarrow T \ | \ E + TE→T ∣ E+T
* T→F ∣ T∗FT \rightarrow F \ | \ T \* FT→F ∣ T∗F
* F→(E) ∣ idF \rightarrow (E) \ | \ idF→(E) ∣ id

Task: Simulate a shift-reduce parser for the input string id + id \* id. Trace the steps of the parser (shift and reduce operations) and check whether the string is accepted by the grammar.

Answer 4

The question and explain how to implement a recursive descent parser for the given grammar in Python, then apply it to check if the input strings are valid expressions.

**Problem Analysis**

You’re given a **context-free grammar (CFG)** that defines a set of valid arithmetic expressions. The CFG rules provided are:

1. **Grammar Rules:**
   * E→T ∣ E+T
   * T→F ∣ T∗F
   * F→(E) ∣ id
2. **Non-terminal Symbols:**
   * E: Represents an expression.
   * T: Represents a term.
   * F: Represents a factor.
3. **Terminal Symbols:**
   * id: An identifier (like a variable or number).
   * +: Addition operator.
   * \*: Multiplication operator.
   * ( and ): Parentheses.
4. **Goal**:
   * Implement a **recursive descent parser** for this grammar, which will check if a given input string matches the grammar rules.
   * Test it on the input strings:
     1. id + id \* id
     2. id \* (id + id)

**Recursive Descent Parsing**

A recursive descent parser is a type of **top-down parser** where each non-terminal symbol in the grammar has a corresponding function. This parser will attempt to match the input tokens to the expected patterns by "calling" each function according to the grammar rules.

**Steps to Implement the Parser**

1. **Define Functions for Each Non-terminal Symbol**:
   * Each function (E, T, and F) will attempt to parse the input string based on the grammar rules.
2. **Define the Matching and Token Functions**:
   * We need a function to match tokens (match) and advance through the input.
   * We also need to keep track of the current position in the input string to parse it correctly.
3. **Implementation of Grammar Rules**:
   * Use recursion to handle nested expressions (like in parentheses or chaining of operations).
4. **Recursive Functions**:
   * E(): Implements E→T ∣ E+T. It first tries to parse a T. If there is a +, it recursively calls T to handle expressions like E + T.
   * T(): Implements T→F ∣ T∗F. Similar to E, but for multiplication.
   * F(): Implements F→(E) ∣ id. It matches either an id (identifier) or recursively an expression enclosed in parentheses (E).
5. **Matching Function**:
   * match(expected\_token): Checks if the current token matches the expected\_token. If it does, it advances to the next token.

**Running the Code**

Given the inputs:

1. id + id \* id
2. id \* (id + id)

**Expected Output**

The code will print:

plaintext

Copy code

Expression id + id \* id: Valid expression

Expression id \* ( id + id ): Valid expression

Both expressions are valid according to the grammar rules.

**Explanation of Each Input**

* **id + id \* id**:
  + This string matches E → E + T where E parses id and T parses id \* id.
  + id \* id is parsed as T → T \* F, where T and F both parse id.
* **id \* (id + id)**:
  + This string matches T → F \* F, where the first F parses id and the second F parses (E).
  + (id + id) matches E → E + T inside the parentheses, with E parsing id and T parsing id.

**Summary**

This recursive descent parser successfully parses expressions based on the grammar provided, handling operations and precedence correctly. The function definitions E, T, and F follow the rules of the grammar and check if each input string matches the grammar rules for a valid expression.

Question 5

Words and Vector

Task: Implement a Python program to:

1. Generate Word Vectors: Use a simple dataset of sentences to create word vectors using the Word2Vec algorithm from the gensim library.
2. Find Similar Words: For a given word, find the top 5 most similar words based on the generated word vectors.
3. Calculate Cosine Similarity: Compute the cosine similarity between two specific word vectors.

Answer 5

from gensim.models import Word2Vec

from sklearn.metrics.pairwise import cosine\_similarity

import numpy as np

# Sample dataset

sentences = [

["I", "am", "going", "to", "the", "market"],

["I", "will", "buy", "apples", "and", "bananas"],

["Tomorrow", "is", "a", "sunny", "day"],

["We", "will", "go", "to", "the", "park"],

["I", "like", "eating", "fresh", "apples"]

]

# Train Word2Vec model on the dataset

model = Word2Vec(sentences, vector\_size=100, window=5, min\_count=1, workers=4)

# Function to find similar words

def find\_similar\_words(word, top\_n=5):

try:

similar\_words = model.wv.most\_similar(word, topn=top\_n)

return similar\_words

except KeyError:

return f"The word '{word}' does not exist in the vocabulary."

# Function to calculate cosine similarity between two words

def cosine\_similarity\_between\_words(word1, word2):

try:

vector1 = model.wv[word1]

vector2 = model.wv[word2]

similarity = cosine\_similarity([vector1], [vector2])

return similarity[0][0]

except KeyError as e:

return f"One of the words '{word1}' or '{word2}' does not exist in the vocabulary."

# Testing the functions

print("Top 5 words similar to 'apples':")

print(find\_similar\_words("apples"))

print("\nCosine similarity between 'apples' and 'bananas':")

print(cosine\_similarity\_between\_words("apples", "bananas"))

output

Top 5 words similar to 'apples':

[('is', 0.25290459394454956), ('the', 0.17016050219535828), ('Tomorrow', 0.15028144419193268), ('eating', 0.13887979090213776), ('day', 0.10852647572755814)]

Cosine similarity between 'apples' and 'bananas':

0.09936432