Language Model -
Grammar Based LM ; Statistical LM.

A language model (LM) is a probabilistic model that determines the likelihood of a sequence of words occuring in a given language.

It helps in generating, predicting, & analyzing text based on observed linguistic patterns. - generate human-like text.

Mathematically, given a sequence of words $W = (w_1, w_2 \ldots w_n)$, a language model estimates the probability:

$$P(W) = P(w_1, w_2, \ldots w_n)$$

By using different techniques (rule-based, statistical, or neural), LMs can predict the next word, complete sentences, or even generate entire documents.

Types of LM :-

Two Major types

→ Grammar-Based LM
→ Statistical LM.

The Importance Of LMs in NLP Tasks :-

Language Models are crucial in NLP because they:

1. Enable machines to understand and generate human language.
2. Improve the performance of tasks like translation, summarization, sentiment analysis, & more.
3. Enhance user interaction in applications like chatbots & virtual assistants.

Grammar - Based Language model.

A grammar-based language model uses predefined set of grammatical rules to construct sentences.

components :-

1. Grammar Rules : Define syntactic structures (eg noun phrase, verb phrase etc)
2. Parsing : The process of analyzing a sentence's structure to identify its grammatical components.

Formal Grammars :-

*) Context - free grammars,
*) Context - sensitive grammars.

Context - free Grammars (CFGs) :

These are grammars where the production rules are applied regardless of the context of a non-terminal. They are represented by the notation

$$G_1 = (N, \Sigma, P, S)$$

where,

1. Non-Terminal Symbols (N):
These a placeholders or variables that can be replaced by other symbols (both non-terminal & terminal).

2. Terminal symbols (Σ):
These are the actual symbols or words in the language that cannot be replaced further.

3. Start Symbol (S):
This is the initial non-terminal Symbol from which the production starts

4. Production rules (P):
These rules define how non-terminal symbols can be replaced by combination of terminal & non-terminal symbols.

The Parsing Process:-

The parsing process can be done using either top-down or bottom-up parsing methods.

Top-down parsing:

→ This method starts with start symbol S & tries to rewrite it to match the i/p sentence.

→ It applies production rules to the start symbol & continues expanding non-terminal symbols until the i/p sentence is derived.

Bottom-up parsing:-

→ This method starts with the i/p sentence & tries to construct the parse tree upwards to reach the start symbol S.

→ It identifies the parts of the i/p sentence that match the right-hand side of the production rules & combines them into non-terminal symbols until the start symbol S is reached.

Ex:- (1) Top-Down:

Given the sentence "The cat sat on the mat", generate a parse tree using context-free grammar.

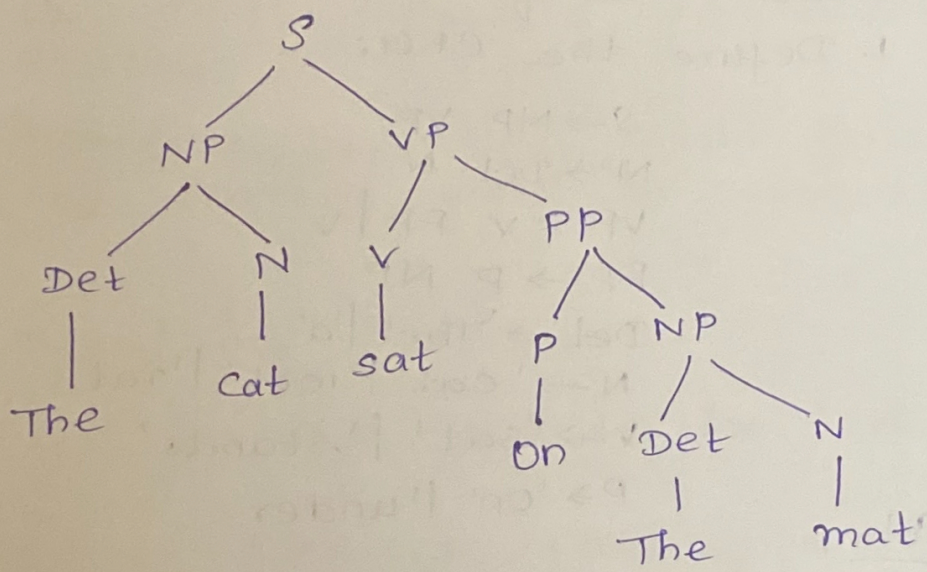
Step-by-step solutions:

1. Define the CFG:

$S \rightarrow NP\ VP$
$NP \rightarrow Det\ N$
$VP \rightarrow V\ PP$
$PP \rightarrow P\ NP$
$Det \rightarrow$ 'The'
$N \rightarrow$ 'cat' | 'mat'
$V \rightarrow$ 'sat'
$P \rightarrow$ 'on'

NP = Noun phrase
VP = verb phrase
PP ⇒ preposition phrase
V = verb;
Det = Determiner (The, a, they, these, them etc)

2. Generate the Parse Tree:



This parse tree shows how the sentence. "The cat sat on the mat" is derived from the start symbol S using the production rules of the CFG.

Ex 2:- (2)

The cat sat on the _____ :

How CFG will generate the correct word for the above sentence.

→ To predict the correct word to complete the sentence "The cat sat on the ____" using CFG, the model would rely on the production rules we've defined earlier.

Production rules for CFG:-

1. Define the CFG:

```
S → NP VP
NP → Det N
VP → V PP | V
PP → P NP
Det → 'The' | 'a'
N → 'cat' | 'mat' | 'rat'
V → 'sat' | 'stands'
P → 'on' | 'under'.
```

Soln :-

Use production rules to predict what comes next (start with S or VP or ...)

(1) Starting with the VP (verb phrase):

```
    V
VP → PP
```

(2) We already have the verb "sat":

```
V → 'sat'
```

(3) The next part should be a Prepositional phrase (PP):

```
PP → P NP.
```

(4.) We have "on" as the preposition:

P → 'on'

(5) Next, we need a Noun phrase (NP)

NP → Det N

(6.) We already have the determiner 'the':

Det → 'the'

(7) finally, we need a noun. The possible options for N are:

N → 'cat' | 'mat' | 'rat'

Given the context, "The cat sat on the mat" is the most natural & grammatically correct sentence.

Therefore, the CFG would generate "mat" as the correct word to complete the sentence.

Resulting Sentence:

The cat sat on the mat.

(3) Example: for Bottom-up parsing :-

It starts with the ilp sentence and gradually works its way up to construct the parse tree until it reaches the start symbol S.

This technique is also known as shift-reduce parsing because it involves shifting ilp symbols onto a stack & reducing them based on production rules.

Let's define our CFG again:

S → NP VP
NP → Det N
VP → V PP | V
PP → P NP
Det → 'The' | 'the' |
N → 'cat' | 'mat'
V → 'sat'
P → 'on'

Parsing steps for "The cat sat on the mat"

1. Input Sentence:

The cat sat on the mat

2. Initialize stack & i/p buffer:

stack: [ ]

i/p buffer: ['The', 'cat', 'sat', 'on', 'the', 'mat']

3. shift i/p symbols onto stack:

a. shift The onto the stack:

stack: ['The']

i/p buffer: ['cat', 'sat', 'on', 'the', 'mat']

b. shift cat onto the stack:-

stack: ['The', 'cat']

i/p buffer: ['sat', 'on', 'the', 'mat']

4. Reduce by production Rule ~~'cat'~~ :
Det → 'The'

stack: [Det, 'cat']

i/p buffer: ['sat', 'on', 'the', 'mat']

5. Reduce by production Rule N → 'cat':

stack : [Det, N]
ip buffer : ['sat', 'on', 'the', 'mat']

6. Reduce by production rule NP → Det N:

stack: [NP]
ip buffer : ['sat', 'on', 'the', 'mat']

7. Shift sat onto the stack:

stack : [NP, 'sat']
ip buffer : ['on', 'the', 'mat']

8. Reduce by production Rule V → 'sat'

stack : [NP, V]
ip buffer : ['on', 'the', 'mat']

9. Shift on onto the stack

stack : [NP, V, 'on']
ip buffer : ['the', 'mat']

10. Reduce by production rule P → 'on':

stack : [NP, V, P]
ip buffer : ['the', 'mat']

11. Shift 'the' onto the stack

stack : [NP, V, P, 'the']
ip buffer : ['mat']

12. Reduce by production rule Det → 'the':

stack : [NP, V, P, Det]
ip buffer : ['mat']

13. Shift mat onto the stack:

stack: [NP, V, P, Det, 'mat']
i/p buffer: []

14. Reduce by production rule: N → 'mat'

stack: [NP, V, P, Det, N]
i/p buffer: []

15. Reduce by production rule NP → Det N

stack: [NP, V, P, NP]
i/p buffer: []

16. Reduce by production rule PP → P NP

stack: [NP, V, PP]
i/p buffer: []

17. Reduce by production rule VP → V PP:

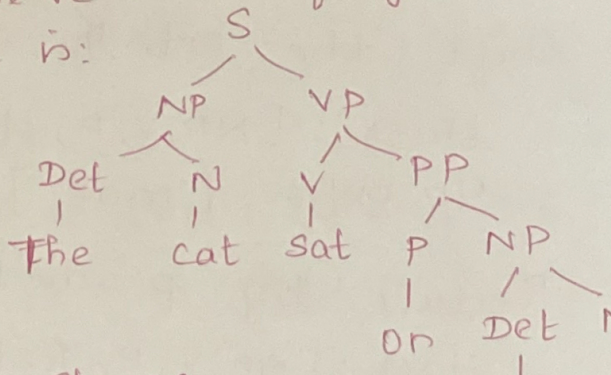
Stack: [NP, VP]
i/p buffer: []

18. Reduce by production rule: S → NP VP

stack: [S]
i/p buffer: []

At this point, we've successfully parsed the sentence.

The final parse tree is:



The parse tree shows how sentence is derived from 'S' using production rules from the bottom-up approach.

Context Sensitive Grammar (CSG)

In these grammars, the production rules are dependent on the context of the non-terminal symbols.

This means that a non-terminal can be replaced by a string if it's surrounded by certain other symbols.

Ex :-

We want to generate

1. The boy runs
2. The boys run using CSG

Non-terminals: S, N_s, N_p, V_s, V_p, Det

Terminals : the, boy, boys, runs, run

Production rule:

$S \rightarrow$ Det N_s V_s

$S \rightarrow$ Det N_p V_p

$N_s \rightarrow$ boy

$N_p \rightarrow$ boys

$V_s \rightarrow$ runs

$V_p \rightarrow$ run

1) $S \rightarrow$ Det N_s V_s

This rule applies when the noun is singular (eg boy) & expect a singular verb (eg runs).

2) $S \rightarrow$ Det Np Vp

This rule applies when the noun is plural (eg boys) & expect plural Verb (eg run)

Ex:- 2 Generate $a^n b^n$, $n \geq 1$ using CFG: with the help of production rule

$S \rightarrow aCb$
$C \rightarrow aCb$
$C \rightarrow A$
$A \rightarrow \varepsilon$ (where ε represents empty string)

Non-terminals: S, A, C
terminals: a, b

Sol:

Let's construct $a^3 b^3$

1. $S \longrightarrow aCb$
2. Apply $C \rightarrow aCb$

   $aCb \rightarrow aacbb$
3. Apply $C \rightarrow aCb$

   $aaCbb \rightarrow aaaCbbb$
4. Apply $C \rightarrow A$

   $aaaCbbb \rightarrow aaaAbbb$
5. Apply $A \rightarrow \varepsilon$

   $aaaAbbb \longrightarrow aaabbb$.

we generate $a^3 b^3$ a string.