Natural Language Processing (TYAIML)

Unit-3

Word-Level Analysis (Statistical Language Models (SLMs)):

Word-Level Analysis in Statistical Language Models (SLMs) refers to the process of analysing and predicting text at the word level using statistical methods. These models estimate the probability of word sequences based on their frequency in a given corpus, enabling tasks like text prediction, machine translation, speech recognition, and spelling correction.

Key Concepts:

N-gram Models-

- The most common approach in statistical language modeling.
- An n-gram is a contiguous sequence of *n* words.
- Examples:
  - Unigram (1-gram): Treats each word independently (e.g., "the", "cat", "sat").
  - Bigram (2-gram): Considers pairs (e.g., "the cat", "cat sat").
  - Trigram (3-gram): Uses triplets (e.g., "the cat sat").
- Probability Estimation:
  - For bigrams, the probability of word given previous word.

Smoothing Techniques-

- Used to handle unseen word combinations (zero-count problem).
- Methods:
  - Laplace (Add-One) Smoothing: Adds 1 to all counts.
  - Good-Turing Discounting: Adjusts frequencies of rare/unseen n-grams.
  - Back off & Interpolation: Uses lower-order n-grams if higher ones are missing.

Perplexity-

A measure of how well a model predicts a sample.

Lower perplexity → better model performance.

Word Prediction & Generation-

- Given a sequence, the model predicts the next word using probabilities.
- Example:
  - Input: "The cat sat on the ___"
  - Model predicts: "mat" (if P(mat|the) is highest).

Applications of Word-Level Statistical Models-

- Autocomplete & Spelling Correction (e.g., Google Search)
- Machine Translation (early statistical MT systems)
- Speech Recognition (predicting likely word sequences)
- Information Retrieval (ranking documents based on word probabilities)

Morphology in NLP

Morphology is the study of the internal structure of words and how they are formed from smaller meaningful units called ***morphemes***. In Natural Language Processing (NLP), morphological analysis is crucial for tasks like text normalization, stemming, lemmatization, and machine translation.

Types of Morphological Processes:

Morphological processes describe how words are formed and modified in a language. These processes are crucial in Natural Language Processing (NLP) for tasks like stemming, lemmatization, and machine translation. Below are the main types:

1. Inflection

Modifies a word to express grammatical features (e.g., tense, number, gender) without changing its core meaning or word class.

Examples:

- Tense: "run" → "ran" (past tense)
- Plurality: "cat" → "cats"
- Possession: "John" → "John's"
- Comparative/Superlative: "fast" → "faster" → "fastest"

Used in NLP for:

- Verb conjugation detection ("goes" → "go")
- Plural-to-singular normalization ("mice" → "mouse")

2. Derivation

Creates new words (often with a meaning shift) by adding prefixes, suffixes, or infixes.

May change the word class (e.g., noun → verb).

Examples:

- "happy" (adj) → "unhappy" (adj, negation)
- "teach" (verb) → "teacher" (noun)
- "logic" (noun) → "illogical" (adj, opposite meaning)

Used in NLP for:

- Expanding vocabulary ("automate" vs. "automation")

- Sentiment analysis ("happy" vs. "unhappy")

3. Compounding

Combines two or more words to form a new word with a combined meaning.

Examples:

- "sun" + "flower" → "sunflower"
- "black" + "board" → "blackboard"
- "tooth" + "brush" → "toothbrush"

Used in NLP for:

- Handling compound nouns in German/Dutch
- Improving machine translation

4. Cliticization

A shortened form where a word attaches to another (often contractions).

Examples:

- "I am" → "I'm"
- "do not" → "don't"

Used in NLP for:

- Normalizing text ("can't" → "cannot")
- Speech recognition ("gonna" → "going to")

5. Reduplication

Repeats all or part of a word to indicate plurality, emphasis, or grammatical function.

Common in Malay, Tagalog, and some African languages.

Examples:

Used in NLP for:

- Handling agglutinative languages
- Morphological segmentation ("kitab" (book) → "kutub" (books) in Arabic)

6. Conversion (Zero Derivation)

Changes a word's grammatical category without any affixation.

Examples:

- "email" (noun) → "to email" (verb)
- "bottle" (noun) → "to bottle" (verb)
- "clean" (adj) → "to clean" (verb)

Used in NLP for:

- POS tagging ("Google" as noun vs. verb)

- Word sense disambiguation

7. Suppletion

Uses completely different word forms for inflection (irregular cases).

Examples:

- "email" (noun) → "to email" (verb)
- "bottle" (noun) → "to bottle" (verb)
- "clean" (adj) → "to clean" (verb)

Used in NLP for:

- Handling irregular verbs in machine translation
- Grammar checking ("he goed" → "he went")

Regular Expression

A Regular Expression (RegEx) is a sequence of special characters and symbols that defines a search pattern primarily used for string matching**,** searching**,** extracting**,** and manipulating text**.** In Natural Language Processing (NLP**)**, RegEx plays a vital role in text preprocessing**.**

Why?

Use Case	Description
Tokenization	Splitting text into words/sentences based on patterns
Cleaning Text	Removing unwanted characters (e.g., punctuation, numbers etc.)
Normalization	Lowercasing, removing repeated characters (e.g., “soooo” → “so”)
Information Extraction	Extracting emails, phone numbers, dates, hashtags, etc.
Pattern Matching	Finding specific words, prefixes, suffixes

Basic RegEx Syntax-

Pattern	Meaning	Example Match
`.`	Any character except newline	`c.t` → cat, cut, c9t
`^`	Start of string	`^Hello` → "Hello world"
`$`	End of string	`world$` → "Hello world"
`*`	0 or more of previous character	`go*` → g, go, goo, gooo
`+`	1 or more of previous character	`go+` → go, goo, gooo
`?`	0 or 1 of previous character	`colou?r` → color, colour
`[]`	Any one character in brackets	`[aeiou]` → a, e, i, o, u
`[^]`	Any one character **not** in brackets	`[^aeiou]` → consonants
`{m,n}`	m to n repetitions	`a{2,4}` → aa, aaa, aaaa
`` ` ``	`` ` ``	OR operator
`()`	Group expressions	`(ha)+` → ha, haha, hahaha
`\d`	Any digit (0-9)	`\d+` → 1, 12, 2025

Pattern	Meaning	Example Match
\w	Word characters (a-z, A-Z, 0-9, _)	\w+ → hello, word_123
\s	Whitespace (space, tab, newline)	\s+ → space, tabs

Examples in NLP-

1. Tokenizing Words and Sentences
   - text = "NLP is fun! Let's learn it."
   - words = re.findall(r'\w+', text)
   - print(words) # ['NLP', 'is', 'fun', 'Let', 's', 'learn', 'it']
2. Removing Punctuation
   - text = "Hello, world!!! NLP is amazing."
   - cleaned = re.sub(r'[^\w\s]', '', text)
   - print(cleaned) # Hello world NLP is amazing
3. Extracting Emails
   - text = "Please contact us at support@example.com or hr@company.org"
   - emails = re.findall(r'\b[\w.-]+@[\w.-]+\.\w{2,4}\b', text)
   - print(emails) # ['support@example.com', 'hr@company.org']
4. Matching Dates
   - text = "My birthday is 12/07/1999 and my friend's is 01-01-2000"
   - dates = re.findall(r'\b\d{2}[-/]\d{2}[-/]\d{4}\b', text)
   - print(dates) # ['12/07/1999', '01-01-2000']
5. Finding Hashtags and Mentions
   - text = "Follow @openai and #AI on Twitter!"
   - hashtags = re.findall(r'#\w+', text)
   - mentions = re.findall(r'@\w+', text)
   - print(hashtags) # ['#AI']
   - print(mentions) # ['@openai']

Application in NLP-

Task	How RegEx Helps
Text preprocessing	Removing special chars, normalizing case
Entity extraction	Emails, dates, phone numbers
Rule-based classification	If tweet contains "urgent
Spelling correction	Detecting repeated characters (e.g., "soooo" → "so")
Grammar checking	Finding repeated punctuation or spacing issues

Advantages-

- Fast and efficient for simple pattern matching
- Works well for preprocessing
- Integrates easily with Python (via `re` module)

Limitations

- Not ideal for complex grammar or deep linguistic tasks
- Hard to maintain with complex rules
- Can become unreadable for long patterns

Morphological Models

A morphological model in Natural Language Processing (NLP) is a computational approach that analyses and processes the structure of words by breaking them down into their smallest meaningful units called morphemes. These models help NLP systems understand how words are formed, inflected, and derived in different languages.

Key Components of Morphological Models:

1. Morphemes: The smallest units of meaning (e.g., "un-" + "happy" + "-ness" → "unhappiness").
2. Stems/Roots: The core meaning-bearing part of a word (e.g., "run" in "running").
3. Affixes: Prefixes, suffixes, and infixes that modify meaning (e.g., "-ing", "un-", "-s").
4. Inflectional Morphology: Changes word form for grammar (e.g., "run" → "runs", "running").
5. Derivational Morphology: Creates new words (e.g., "happy" → "happiness").

Types of Morphological Models:

1. Rule-Based Models-

- Use handcrafted linguistic rules for morphological analysis.
- Example: Finite-State Transducers (FSTs) for stemming and lemmatization.
- Pros: Precise for well-defined languages.
- Cons: Hard to scale for irregular or agglutinative languages.

2. Statistical Models-

- Learn morphology from data using probabilities.
- Examples:
    - Morfessor (unsupervised segmentation)
    - Hidden Markov Models (HMMs) for POS tagging
- Pros: Adaptable to different languages.

- Cons: Requires large datasets.

3. Neural Morphological Models-

- Use deep learning to learn word structure.
- Examples:
    - LSTMs & Transformers for morphological inflection (e.g., "go" → "went").
    - Byte-Pair Encoding (BPE) & WordPiece (used in BERT, GPT).
- Pros: Handles rare and complex words well.
- Cons: Computationally expensive.

Applications of Morphological Models:

- Stemming & Lemmatization (reducing words to base forms)
- Morphological Parsing (breaking words into morphemes)
- Machine Translation (handling word forms across languages)
- Spell Checking & Grammar Correction
- Handling Out-of-Vocabulary (OOV) Words

Challenges in Morphological Modeling:

- Morphological Richness (e.g., Turkish, Finnish have complex word structures)
- Ambiguity (e.g., "running" = verb or noun?)
- Irregular Forms (e.g., "go → went", "mouse → mice")
- Low-Resource Languages (lack of training data)

Theory of Computation (ToC):

The Theory of Computation is a branch of computer science that studies the fundamental capabilities and limitations of computers. It explores what problems can be solved, how efficiently they can be solved, and what kinds of abstract machines are needed to solve them.

Important concepts in ToC:

1. Symbol

- Definition: The smallest, indivisible unit in a formal system.
- Role: Building block for strings and languages.
- Examples:
    - Binary: 0, 1
    - English: a, b, c
    - Programming: +, -, (

2. Alphabet (Σ)

- Definition: A finite, non-empty set of symbols.

- Properties:
    - Fixed for a given system (e.g., $\Sigma = \{a, b\}$).
    - Size = $|\Sigma|$ (e.g., $|\{0,1\}| = 2$).
- Examples:
    - Binary: $\Sigma = \{0, 1\}$
    - DNA: $\Sigma = \{A, T, C, G\}$

3. String (or Word)

- Definition: A finite sequence of symbols from an alphabet.
- Key Terms:
    - Length: Number of symbols (e.g., "101" has length 3).
    - Empty String (ε): String with 0 symbols.
- Examples: Over $\Sigma = \{a, b\}$: "aab", "ba", ε

4. Language

- Definition: A set of strings chosen from Σ^*.
- Types:
    - Finite Language: Limited strings (e.g., {ε, "a", "aa"}).
    - Infinite Language: Unlimited strings (e.g., {"a", "aa", "aaa", ...}).
- Examples:
    - All binary strings with even 0s: {ε, "1", "00", "1001", ...}
    - Valid Python identifiers: {"x", "count", "_tmp", ...}

5. Kleene Star (Σ^*)

- Definition: The set of all possible strings (including ε) over Σ.
- Formula: $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup ...$ (where $\Sigma^0 = \{\varepsilon\}$).
- Example: If $\Sigma = \{a\}$, then $\Sigma^* = \{\varepsilon, "a", "aa", "aaa", ...\}$.

6. Power of Σ (Σ^n)

- Definition: All strings of exactly length n over Σ.
- Example: $\Sigma = \{0,1\}$, $\Sigma^2 = \{"00", "01", "10", "11"\}$.

7. Automaton (Automata)

- Definition: An abstract mathematical model to recognize languages.
- Purpose:
    - Accept/Reject strings based on rules.
    - Model real-world machines (e.g., compilers, vending machines).

Finite State Morphology (FSM)

Finite State Morphology (FSM) is a rule-based, computational approach to modeling word structure using finite-state automata (FSA) and transducers (FST). It is widely used in NLP for tasks like morphological analysis, stemming, and spell-checking.

Core Concepts of FSM

(A) Finite-State Automata (FSA)

- Purpose: Recognize whether a word belongs to a language.
- Example: Language: English plural nouns (cat → cats, dog → dogs).

  FSA: Accepts words ending with -s.

(B) Finite-State Transducers (FST)

- Purpose: Map between surface forms (actual words) and lexical forms (stems + morphemes).
- Example: Input: "running" → Output: run + V + PRES_PART.

  Encodes morphological rules (e.g., run + ing → running).

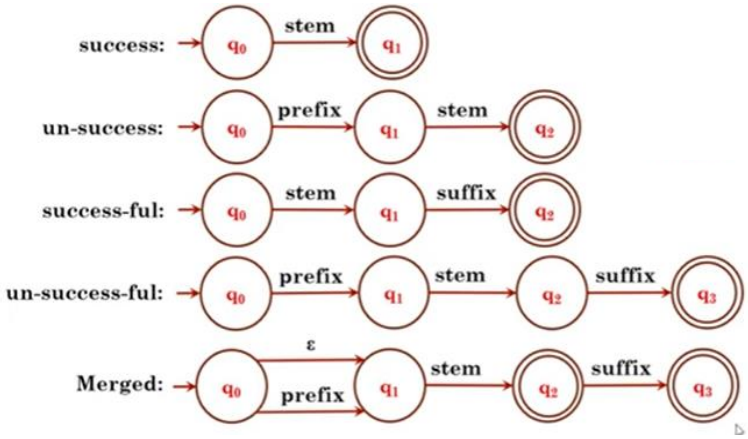
How FSM Works

Step 1: Morphological Analysis

- Input: Surface form (e.g., "unhappiness").
- FST Decomposition:

Text- "unhappiness" → un + happy + ness

- Tags Added:
  - un- (prefix: negation)
  - happy (root)
  - -ness (suffix: noun-forming).



Step 2: Morphological Generation

- Input: Lexical form (e.g., happy + ADJ + COMP).
- FST Output: "happier".

Key Components of FSM

Component	Role	Example
Lexicon	Database of stems/affixes.	{run, happy, un-, -ness, -ing}
Morphotactics	Rules for morpheme ordering.	un- must precede adjectives.
Orthographic Rules	Handles spelling changes.	run + ing → running (double ‘n’).
FST Cascade	Transducers for analysis/generation.	Tokenizer → Stemmer → Tagger.

Applications of FSM:

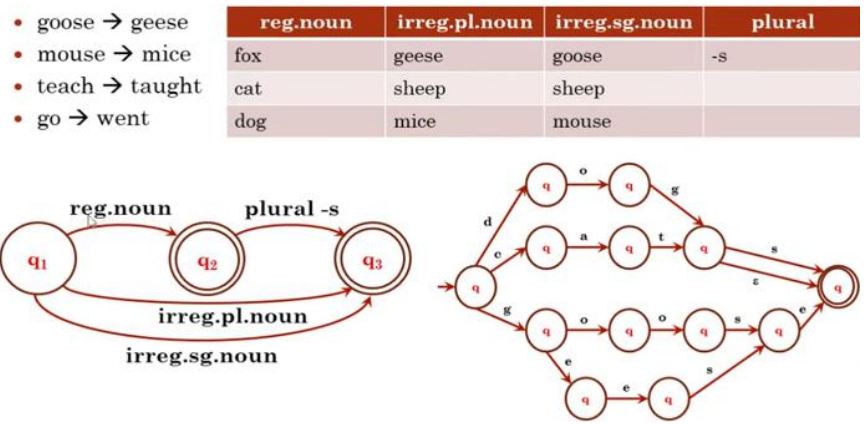
1. Stemming & Lemmatization
    - Reduces "running" → "run" (Porter Stemmer uses FSM-like rules).
2. Spell Checking
    - Uses FSTs to suggest corrections (e.g., "hapy" → "happy").
3. Machine Translation
    - Analyzes word structure before translation.
4. Morphological Tagging
    - Labels words with features (e.g., cat + N + PL → "cats").

Advantages of FSM:

- Efficiency: Fast lookup (used in real-time apps like spell-checkers).
- Precision: Handles complex morphology (e.g., Finnish, Turkish).
- Deterministic: No ambiguity in analysis (unlike statistical methods).

Limitations of FSM:

- Manual Effort: Requires linguistic expertise to write rules.
- Inflexible: Struggles with irregular forms (e.g., "go" → "went").
- Scalability: Hard to maintain for highly agglutinative languages.



Deterministic Finite Automata (DFA)

A Deterministic Finite Automaton (DFA) is a theoretical machine used to recognize regular languages (patterns defined by regular expressions). It's a foundational concept in automata theory and compiler design.

Formal Definition of DFA:

A DFA is a 5-tuple:

$M = (Q, \Sigma, \delta, q_0, F)$

- Q: Finite set of states
- Σ: Input alphabet (finite set of symbols)
- δ: Transition function ($Q \times \Sigma \rightarrow Q$)
- q_0: Initial state ($q_0 \in Q$)
- F: Set of accepting/final states ($F \subseteq Q$)

Key Properties of DFA:

- Deterministic: Exactly one transition per symbol in each state.
- No ε-moves: Cannot change state without reading input.
- Finite Memory: Only knows current state (no stack/tape).

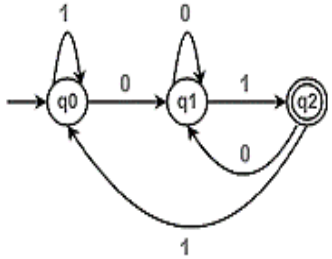
How a DFA Works:

1. Starts in initial state q_0.
2. Reads input string one symbol at a time.
3. Follows δ(current_state, input_symbol) to next state.
4. After last symbol:
    - If current_state ∈ F → Accept
    - Else → Reject

Example of DFA: DFA for "Strings Ending with 01"

Language: All binary strings ending with "01".

State Diagram-



- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0, 1\}$
- $F = \{q_2\}$

Transition Table-

State	Input 0	Input 1
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_1	q_0

Sample Runs

- Input "0101":

  $q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_1 \rightarrow q_2$ (Accept)
- Input "110":

  $q_0 \rightarrow q_0 \rightarrow q_0 \rightarrow q_1$ (Reject)

Applications of DFA:

1. Lexical Analysis (Token recognition in compilers)
    - Example: Identifiers (/[a-zA-Z_][a-zA-Z0-9_]*/).
2. Text Search (grep, regex engines optimize to DFAs).
3. Hardware Design (Digital circuit state machines).
4. Network Protocols (Packet filtering rules).

Limitations of DFA:

1. Only Regular Languages: Cannot handle nested structures like $a^n b^n$.
2. State Explosion: Complex patterns may need exponentially many states.

More illustrations on DFA:

1. Construct a DFA which accept a language of all strings starting with 'a' over { a, b }.

L_1={a, aa, aaa, ab, abb, aba, aab.....}

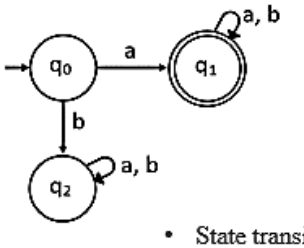
States:

q_0: Initial state (start).
q_1: Accepting state (strings start with 'a').
q_2: Dead/trap state (strings that do not start with 'a').

Transitions:

From q_0:
On input 'a', go to q_1 (valid start).
On input 'b', go to q_2 (invalid start).
From q_1 (accepting state):
On input 'a' or 'b', stay in q_1 (any continuation is allowed).
From q_2 (dead state):
On input 'a' or 'b', stay in q_2 (once invalid, always invalid).

- State transition diagram

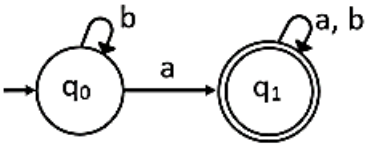


- State transition table

State	Input 'a'	Input 'b'
q_0	q_1	q_2
q_1	q_1	q_1
q_2	q_2	q_2

2. Construct a DFA which accept a language of all strings containing 'a' over { a, b }.

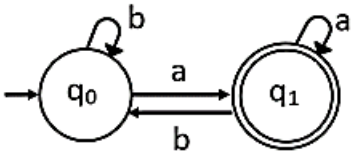
L_2={a, aa, aaa, ab, abb, aba, aab.....}



State	Input 'a'	Input 'b'
q_0	q_1	q_0
q_1	q_1	q_1

3. Construct a DFA which accept a language of all strings ending with 'a' over { a, b }.

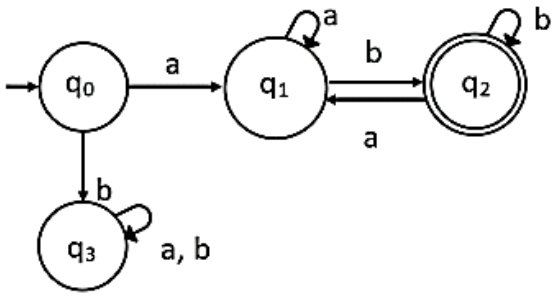
L_3={a, aa, aaa, ba, abba, aba, aabaa.....}



State	Input 'a'	Input 'b'
q_0	q_1	q_0
q_1	q_1	q_0

4. Construct a DFA which accept a language of all strings starting with 'a' and ending with 'b' over { a, b }.

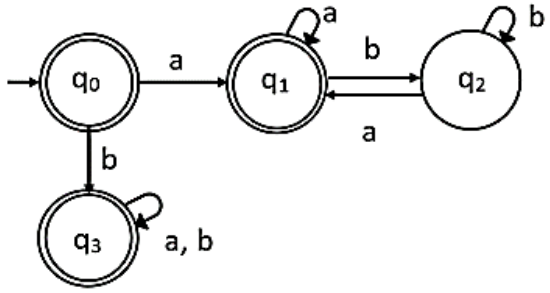
L_4={ab, aab, abab, abb, aabb, abbab, ababb……}



State	Input 'a'	Input 'b'
q_0	q_1	q_3
q_1	q_1	q_2
q_2	q_1	q_2
q_3	q_3	q_3

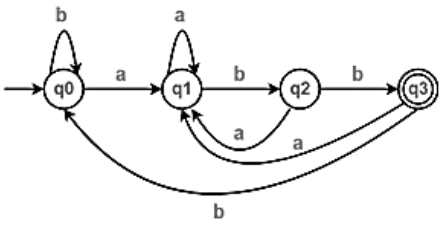
5. Construct a DFA which accept a language of all strings not starting with 'a' or not ending with 'b' over { a, b }. L={w ϵ {a,b} | w not start with a and not end with b}

L_5={ba, baba, bbaa, baa, bbaaa……}

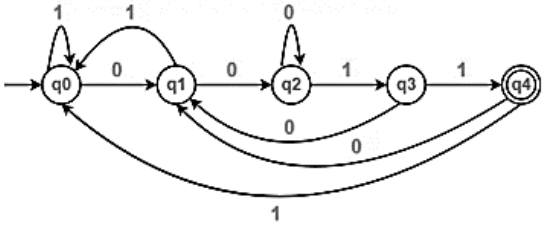


State	Input 'a'	Input 'b'
q_0	q_1	q_3
q_1	q_1	q_2
q_2	q_1	q_2
q_3	q_3	q_3

6. Draw a DFA for the language accepting strings ending with 'abb' over {a, b}



7. Draw a DFA for the language accepting strings ending with '0011' over {0, 1}



Non-Deterministic Finite Automata (NFA)

A Non-Deterministic Finite Automaton (NFA) is a finite automaton that generalizes DFAs by allowing multiple possible transitions from a state for the same input symbol, including ε-transitions (empty moves). NFAs recognize the same class of languages as DFAs (regular languages) but often with fewer states.

Formal Definition of NFA

An NFA is a 5-tuple:

$M = (Q, \Sigma, \delta, q_0, F)$

- Q: Finite set of states
- Σ: Input alphabet (including ε for some NFAs)
- δ: Transition function $(Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(Q)$ where P(Q) is the power set of Q
- q_0: Initial state ($q_0 \in Q$)
- F: Set of accepting/final states ($F \subseteq Q$)

Key Differences from DFA

- Multiple transitions allowed for the same symbol.
- ε-transitions allow state changes without reading input.
- Non-deterministic: At any step, the machine can "guess" the correct path.

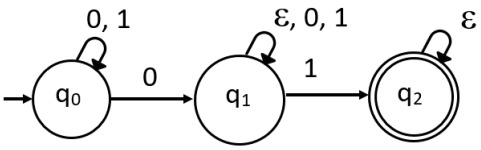
How an NFA Works

1. Starts in initial state q_0 (and all states reachable via ε-moves).
2. For each input symbol, it non-deterministically follows all possible transitions.
3. If any path ends in an accepting state after processing the entire input → Accept.
4. Else → Reject.

Example: NFA for "Strings Ending with 01"

Language: All binary strings ending with "01".

State Diagram-



- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0, 1\}$
- $F = \{q_2\}$
- δ:

State	Input 0	Input 1	Input ε
q_0	$\{q_0, q_1\}$	$\{q_0\}$	∅
q_1	∅	$\{q_2\}$	$\{q_0\}$
q_2	∅	∅	$\{q_1\}$

Sample Runs

- Input "01" (Accept):
  - Path 1: $q_0 \rightarrow q_0 \rightarrow q_1 \rightarrow q_2$
  - Path 2: $q_0 \rightarrow q_1 \rightarrow q_2$
- Input "00" (Reject):
  - Only path: $q_0 \rightarrow q_0 \rightarrow q_0$

NFA vs. DFA

Feature	DFA (Deterministic Finite Automata)	NFA (Non-deterministic Finite Automata)

Equivalence of NFA and DFA

- Any NFA can be converted to an equivalent DFA using subset construction.
- The DFA states represent sets of NFA states.
- Example: NFA with n states → DFA with up to 2^n states.

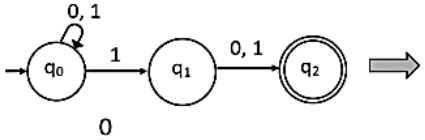
Subset Construction Steps

1. Start with ε-closure of q_0 as the initial DFA state.
2. For each symbol, compute transitions by taking ε-closure of all reachable NFA states.
3. Repeat until no new states are generated.
4. Final states in DFA are those containing any NFA final state.

Construct a NFA which accept a language of all binary strings in which second last bit is '1'.

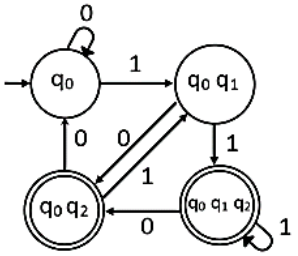
L_S={10, 11, 010, 110, 111, 11110.....}

NFA



State	Input '0'	Input '1'
q_0	q_0	q_0q_1
q_1	q_2	q_2
q_2	-	-

DFA



State	Input '0'	Input '1'
q_0	q_0	q_0q_1
q_0q_1	q_0q_2	$q_0q_1q_2$
q_0q_2	q_0	q_0q_1
$q_0q_1q_2$	q_0q_2	$q_0q_1q_2$

Applications of NFAs

1. Regex Engines (Perl, Python re module first convert to NFA).
2. Lexical Analysis (Flex/Lex use NFA-based pattern matching).
3. Model Checking (Formal verification of systems).

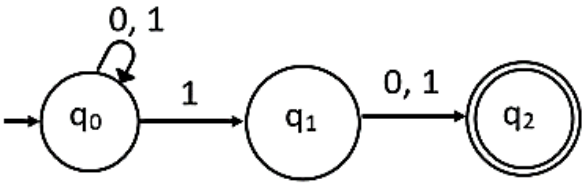
Limitations of NFAs

1. Slower execution than DFAs (due to non-determinism).
2. No practical direct hardware implementation (unlike DFAs).

More illustrations on NFA:

1. Construct a NFA which accept a language of all binary strings in which second last bit is '1'.

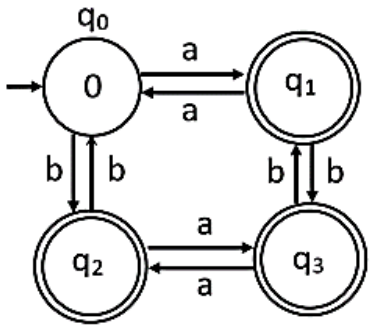
   L_1={10, 11, 010, 110, 111, 11110.....}



State	Input '0'	Input '1'
q_0	q_0	q_0q_1
q_1	q_2	q_2
q_2	-	-

2. Construct a NFA which accept a language of all strings that has even number of 'a's and even number of 'b's.

   L_2={ϵ, aa, bb, abab, aabb, bbaa, baba, abba, baab.....}



State	Input 'a'	Input 'b'
ϵ	-	-
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_3	q_0

- If q_1 is the only accepting state then odd a and even b
- If q_2 is the only accepting state then even a and odd b
- If q_3 is the only accepting state then odd a and odd b

Finite State Transducers

What is a Finite State Transducer (FST)?

At its core, an FST is a type of Finite State Automaton (FSA) that doesn't just accept or reject a sequence of symbols (like a word); it also produces an output sequence.

Think of it as a state machine with two tapes:

1. An input tape (e.g., the word you read).
2. An output tape (e.g., the transformed word you write).

As the FST reads each symbol from the input tape, it moves between states and writes zero, one, or more symbols to the output tape. This makes it a powerful model for tasks that involve mapping or transforming one string to another.

Formal Definition

An FST is a 6-tuple $(Q, \Sigma, \Gamma, I, F, \delta)$ where:

- Q is a finite set of states.
- Σ is a finite input alphabet.
- Γ is a finite output alphabet.
- $I \subseteq Q$ is the set of initial states.
- $F \subseteq Q$ is the set of final (accepting) states.
- $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \times Q$ is the transition relation. A transition (q, a, b, r) means that from state q on input a, the machine moves to state r and outputs b. ε represents the empty string.

How do FSTs work?

Let's create a simple FST for English pluralization. The rule is: "Add '-s' to the end of a noun, unless it ends in s, x, z, ch, or sh, in which case add '-es'."

This is complex for a simple regex, but an FST can handle it elegantly.

Input: cat

Desired Output: cats

A simplified FST for this might have a path that reads c, outputs c; reads a, outputs a; reads t, outputs t; and then, seeing the end of the word, follows a special transition that reads nothing (ε) and outputs s.

Input: box

Desired Output: boxes

The FST would read b and output b, read o and output o, read x and output x. Then, because the last letter was x, it would follow a different final transition that reads nothing (ε) and outputs es.

The power comes from the FST's ability to "remember" what it just saw (its current state) to decide what to output next.

Key Properties of FSTs

1. Bidirectionality: FSTs can run in reverse. You can generate the input from the output, which is very useful for analysis.

2. Composition: You can combine two or more FSTs into a single, more complex transducer. For example, you could have one FST for spelling correction and another for stemming, then compose them into a single operation.
3. Determinizability: Some FSTs can be made deterministic, meaning for any given state and input symbol, there is at most one possible transition. This makes them very fast to execute.
4. Weighted FSTs: This is a crucial extension. You can add weights (probabilities or costs) to transitions. This allows the FST to represent a probability distribution over possible outputs, which is fundamental in modern NLP. The Viterbi algorithm is used to find the most likely path (and thus the most likely output) through a weighted FST.

Applications of FSTs in NLP

Application	Description	Example
Morphological Analysis	Breaking a word down into its morphemes (root + affixes) and determining its grammatical features.	Input: unbelievably Output: un + believe + ably + ADV
Morphological Generation	The inverse of analysis. Generating the surface form of a word from its lemmatized form and features.	Input: believe + VERB + PAST Output: believed
Part-of-Speech (POS) Tagging	Assigning grammatical tags to words. A simple POS tagger can be built by cascading a morphological analyzer FST with a disambiguation step.	Input: The cat sat Output: The/DET cat/NOUN sat/VERB
Spell Checking & Correction	Identifying and correcting misspelled words. An FST can model common errors (insertions, deletions, substitutions, transpositions) with costs.	Input: acress Possible Outputs: actress (cost: 1), acres (cost: 1)
Text Normalization	Converting text into a consistent, canonical form. This is vital in Speech-to-Text and Text-to-Speech systems.	Input: I owe $5.50 to Dr. Smith. Output: I owe five dollars and fifty cents to doctor smith.
Phonology and Grapheme-to-Phoneme	Converting written text (graphemes) into pronunciation (phonemes).	Input: tomato Output: təmeɪɾoʊ (US pronunciation)

Morphological Parsing using FST

- Analyzing word forms and generating inflected forms (e.g., "run" → "running", "ran").
- Mapping between surface forms and lexical forms.
- e.g. To map surface forms (inflected word forms like "cats") to lexical forms (root + grammatical features like "cat +N +PL").
- Input: Accepts surface word forms like cats
- Output: Produces corresponding lexical forms like cat+N+PL
  - So for the word "cats", transitions might look like:
  - c:c → matches character c
  - a:a → matches a
  - t:t → matches t
  - s:ε → consumes plural marker s and maps to an output like +PL at a final state

Input Word	FST Output
cats	cat +N +PL
cat	cat +N +SG (implied)

Design a FST with E-insertion orthographic rule that parses from surface level "foxes" to lexical level "fox+N+PL" using FST.

Step 1: Morphological rule

Plural formation in English (simplified): If a noun ends in sibilant (s, z, sh, ch, x), we add -es instead of just -s.

Rule:+PL → es / {s, z, x, sh, ch}#(i.e., when the plural morpheme follows one of those endings, insert an epenthetic "e").

So, fox + N + PL → foxes

Step 2: Input/Output representations

Lexical level: f o x +N +PL

Surface level: f o x e s

Step 3: FST States & Transitions

We need an FST that maps lexical string → surface string.

(a) Root mapping: f → f, o → o, x → x

(b) Morpheme boundary & category tags:

+N → ε (tags don't appear in surface form),

+PL → es (plural suffix with epenthetic "e" after sibilant x)

(c) Rule handling:

Instead of directly +PL → s, we have conditional rule: If preceding segment ∈ {s, z, x, sh, ch}, output "es".Otherwise, output "s".

Final output: foxes, Final lexical trace: fox+N+PL

Step 4: FST sketch:

We can illustrate as lexical → surface transitions:
(q0) --f:f--> (q1),
(q1) --o:o--> (q2),
(q2) --x:x--> (q3),
(q3) --+N:ε--> (q4),
(q4) --+PL:es--> (qf),

Step 5: Generalization

This FST works not only for foxes but for other sibilant nouns:
bus+N+PL → buses
church+N+PL → churches

N-grams in NLP

N-grams are contiguous sequences of 'n' items from a given text or speech. In Natural Language Processing (NLP), these items are typically words, but they can also be characters or syllables.

Types of N-grams:

- Unigram (1-gram): Single words
- Bigram (2-gram): Pairs of consecutive words
- Trigram (3-gram): Triplets of consecutive words
- 4-gram, 5-gram, etc. (Higher-order n-grams)

Mathematical Representation

For a sequence of words: $W = w_1, w_2, w_3, ..., w_n$

- Unigram: $P(w_i)$

- Bigram: $P(w_i | w_{i-1})$
- Trigram: $P(w_i | w_{i-2}, w_{i-1})$

Importance in NLP:

1. Capturing Context

   Traditional bag-of-words models lose word order information. N-grams preserve local context and word sequences.

2. Language Modeling

   N-grams form the foundation of statistical language models that predict the probability of word sequences.

3. Feature Engineering

   N-grams serve as powerful features for machine learning models in tasks like:

- Text classification
- Sentiment analysis
- Spam detection
- Machine translation

N-gram Probability Calculation-

Maximum Likelihood Estimation (MLE)

- Unigram: P(w) = count(w) \N
- Bigram: $P(w_i | w_{i-1}) = count(w_{i-1}, w_i) \ count(w_{i-1})$
- Trigram: $P(w_i | w_{i-2}, w_{i-1}) = count(w_{i-2}, w_{i-1}, w_i) \ count(w_{i-2}, w_{i-1})$

Example: For the sentence "the quick brown fox"

- Unigrams: ["the", "quick", "brown", "fox"]
- Bigrams: ["the quick", "quick brown", "brown fox"]
- Trigrams: ["the quick brown", "quick brown fox"]

Applications of N-grams-

1. Text Generation
2. Spelling Correction
3. Machine Translation
4. Speech Recognition

Advantages of N-grams:

- Simple and effective for many NLP tasks
- Computationally efficient compared to neural models

- Interpretable - probabilities have clear meanings
- Works well with limited data

Limitations of N-grams::

- Sparsity problem - many n-grams never occur
- Context window limited to n-1 words
- Doesn't capture long-range dependencies
- Storage intensive for large n and large vocabularies

Language Model in NLP

A language model is a system that assigns a probability to a piece of text. Its most fundamental task is to predict the next word in a sequence.

Think about your smartphone's keyboard. When you type "Have a great...", it suggests words like "day," "weekend," or "time." It's using a language model to guess what you're most likely to say next.

Formal Definition: A language model estimates the probability of a sequence of words or characters:

$P(w_1, w_2, w_3, ..., w_n)$

Applications of LM in NLP

1) OCR (Optical character recognition) & Hand recognition

In OCR, LM can be used to predict a word that is not easily readable in the given image.

2) Correcting a sentence

For e.g. "Deer Sir" instead of "Dear Sir"In corpus, probability of “Dear Sir” is more t han “Deer Sir”.

3) Speech Recognition

The sounds for "eye" and "I" are identical.

The probability of the phrase "I am eating" is much greater than the probability of the nonsensical phrase "Eye am eating" (P(I am eating) >> P(Eye am eating)).

Conversely, the probability of the phrase "His eye got hurt" is much greater than the probability of the ungrammatical phrase "His I got hurt" (P(His eye got hurt) >> P(His I got hurt)).

By calculating the probability of word sequences from its training corpus, the N-gram model predicts the correct and most likely word.

4) Machine Translation

The example demonstrates how an N-gram model chooses the most natural-sounding translation from multiple possibilities.

The Hindi word "Bada Bhai" can be translated to the English synonyms "Big Brother" or "Large Brother".

However, the probability of the common phrase "big brother" appearing in the English corpus is much higher than the uncommon phrase "large brother" (P(big brother) >> P(large brother)).

Therefore, the N-gram model selects "Big Brother" as the most probable and appropriate translation.

5) Suggestions while typing

The word prediction may be very helpful for typing. For example, when we type messages on our cell phone, we get suggestions. Suggestions come between the words as well as across a single word. These suggestions make it very easy to type.e.g. “don’t want to…..” [suggest between words]e.g. “close to c….” [suggest across a single word]

6) Auto Completion prediction

In search engine, when we type a query, before completion, search engine gives out various suggestions.For example, query is “how to ….”. After how to, it gives out various suggestions like how to learn computer, how to learn Java, how to learn maths etc.

7) Context Sensitive Spelling Correction

Example: "college is closed on Saturday and Sunday"

Spelling error: The word "college" has been mistakenly typed as "collage".

Definitions:

College: An institution of higher education, similar to a university.

Collage: A collection of different pieces that are put together into one piece of artwork.

The Problem:

Since both "college" and "collage" are valid words present in the dictionary (corpus), a simple spell checker that only checks if a word exists would not flag this as an error.

The Solution:

This requires spelling correction based on local context. An N-gram model can compare the probability of the entire sentence with each word.

The probability of the correct sentence, "college is closed on Saturday and Sunday", is much higher.

The probability of the incorrect sentence, "collage is closed on Saturday and Sunday", is very low, as the word "collage" does not fit the context of being "closed on Saturday and Sunday."

By calculating which word sequence is more probable, the N-gram model can detect and correct this type of context-sensitive spelling error.

Smoothing in NLP.

The Problem: Zero Probability in N-Grams

In language modeling (e.g., unigram, bigram, trigram models), we calculate probabilities of words or sequences of words based on their counts in the training corpus.

The issue is:

- Many valid word sequences (n-grams) might never appear in the training data.
- If an n-gram has count = 0,

Example:

Suppose our training corpus is:

I like cats

I like dogs

From this, we can build a bigram model.

Counts:

- "I like" = 2
- "like cats" = 1
- "like dogs" = 1
- "cats play" = 0 (never appeared in training)

Probabilities (without smoothing):

count (cats play)=0 count(cats)=1

P(cats play) =0/1 = 0

This means if we try to compute the probability of the sentence "I like cats play", the whole sentence probability becomes 0, even though it's a valid English sequence.

This causes problems because:

- Even one zero makes the probability of a whole sentence zero.
- But in reality, the n-gram is possible—it just didn't appear in the training data.

This is catastrophic for a language model:

- Infinite Perplexity: If a single unseen n-gram in a test sentence gets a probability of zero, the entire sentence's probability becomes zero. This leads to a perplexity of infinity, making the model useless for evaluation.
- Poor Generalization: The model will be completely unable to process any new text that contains a novel but reasonable word sequence.

Perplexity:

Perplexity is a way to evaluate how well a language model predicts a test set.

- It measures the "uncertainty" of the model.
- Lower perplexity = better model (less "surprised" by the test data).

Formula-

For a sequence of words $W = w_1, w_2, ..., w_N$:

$$PP(W) = P(w_1, w_2, ..., w_N)^{-\frac{1}{N}}$$

Where:

- N = number of words in the test sentence.
- $P(w_i|w_{i-1})$ = probability from the language model (e.g., bigram).

Example 1 -

Suppose we have a bigram model trained on a small corpus- '<s>I like cats'

Compute probabilities: $P(\text{I}|<s>)=0.5$, $P(like|I)=0.25$, $P(cats|like)=0.125$

Total probability of the sentence: $P(W)=0.5\times0.25\times0.125=0.015625$

$\because\ P(A_1, A_2, ..., A_n) = P(A_1) \cdot P(A_2|A_1) \cdot P(A_3|A_1, A_2) \cdot ... \cdot P(A_n|A_1, A_2, ..., A_{n-1})$

Compute Perplexity

Sentence length N=3.

$$PP(W)=P(W)^{(1/N)}=(0.015625)^{(1/3)}\approx4$$

Interpretation: Perplexity = 4 means, on average, the model is as "confused" as if it had to choose between 4 equally likely words at each step.

Lower perplexity → model is more confident and better at predicting.

Example 2-

Consider the following training data:

<s> I am Henry </s>
<s> I like college </s>
<s> Do Henry like college </s>
<s> Henry I am </s>
<s> Do I like Henry </s>
<s> Do I like College </s>
<s> I do like Henry </s>

Calculate Perplexity for the following using both Bigram and Trigram

1. <s> I like college </s>
2. <s> Do I like Henry </s>

- Unigram counts (relevant ones):
  `<s>:7, </s>:7, I:6, like:5, Henry:5, Do:3, college:2, College:1, am:2, do:1`
- Bigram counts (relevant):
  `(<s>,I):3, (I,like):3, (like,college):2, (college,</s>):2, (<s>,Do):3, (Do,I):2, (like,Henry):2, (Henry,</s>):3`
- Trigram counts (relevant):
  `(<s>,I,like):1, (I,like,college):1, (like,college,</s>):2, (<s>,Do,I):2, (Do,I,like):2, (I,like,Henry):1, (like,Henry,</s>):2`

1) `<s> I like college </s>`

Bigram probabilities

- $P(I|<s>) = 3/7 = 0.4286$
- $P(like|I) = 3/6 = 0.5$
- $P(college|like) = 2/5 = 0.4$
- $P(</s>|college) = 2/2 = 1.0$

Perplexity (N=4):

$$\text{PP}_{\text{bigram}} \approx \mathbf{1.848}$$

Trigram probabilities

- $P(like|<s>, I) = 1/3 = 0.333$
- $P(college|I, like) = 1/3 = 0.333$
- $P(</s>|like, college) = 2/2 =$

Perplexity (N=3):

$$\text{PP}_{\text{trigram}} \approx \mathbf{2.080}$$

2) `<s> Do I like Henry </s>`

Bigram probabilities

- $P(Do|<s>) = 3/7 = 0.4286$
- $P(I|Do) = 2/3 = 0.6667$
- $P(like|I) = 3/6 = 0.5$
- $P(Henry|like) = 2/5 = 0.4$
- $P(</s>|Henry) = 3/5 = 0.6$

Perplexity (N=5):

$$\text{PP}_{\text{bigram}} \approx \mathbf{1.963}$$

Trigram probabilities

- $P(I|<s>, Do) = 2/3 = 0.6667$
- $P(like|Do, I) = 2/2 = 1.0$
- $P(Henry|I, like) = 1/3 = 0.3333$
- $P(</s>|like, Henry) = 2/2 = 1.0$

Perplexity (N=4):

$$\text{PP}_{\text{trigram}} \approx \mathbf{1.456}$$

Smoothing

Smoothing is the solution to this problem. It is a set of techniques designed to adjust the probability estimates so that no n-gram has a probability of exactly zero.

The Core Idea of Smoothing

All smoothing techniques follow the same fundamental principle:

"Steal" a little bit of probability mass from the seen events (n-grams with high counts) and "give" it to the unseen events (n-grams with zero counts).

This ensures that every possible n-gram has at least a tiny, non-zero probability.

Common Smoothing Techniques

Here are some of the most important smoothing methods, from simple to advanced.

1. Laplace Smoothing (Add-One Smoothing)

This is the simplest technique. You add 1 to the count of every n-gram (even unseen ones).

Original MLE (Maximum Likelihood Estimate):

$$P(n\text{-gram}) = \frac{count(n\text{-gram})}{N}$$

Laplace (Add-One) Version:

$$P_{Laplace}(w) = \frac{count(w) + 1}{N + V}$$

where V is the size of the vocabulary (the number of unique words).

Problem: This is a very blunt instrument. It adds too much probability to unseen events, drastically distorting the probabilities of common n-grams. It's rarely used in practice for anything but the simplest demonstrations.

Example 1-

Suppose vocabulary V={cat, dog, lion}, so V=3.

$$P_{Laplace}(w) = \frac{count(w) + 1}{N + V}$$

Corpus: cat dog cat cat dog

Counts: cat = 3 dog = 2 lion = 0 Total words N=5

1) Without smoothing (MLE)

P(cat)=3/5=0.6,

P(dog)=2/5=0.4,

P(lion)=0/5=0

Problem: unseen word "lion" gets probability = 0.

2) With Laplace (Add-One) smoothing

Here N=5, V=3, denominator = 5+3=8.

P(cat)=(3+1)/8=4/8=0.5

P(dog)=(2+1)/8=3/8=0.375

P(lion)=(0+1)/8=1/8=0.125

- *Now no probability is zero.*
- *Even unseen words (like "lion") get a small chance.*

Example 2-

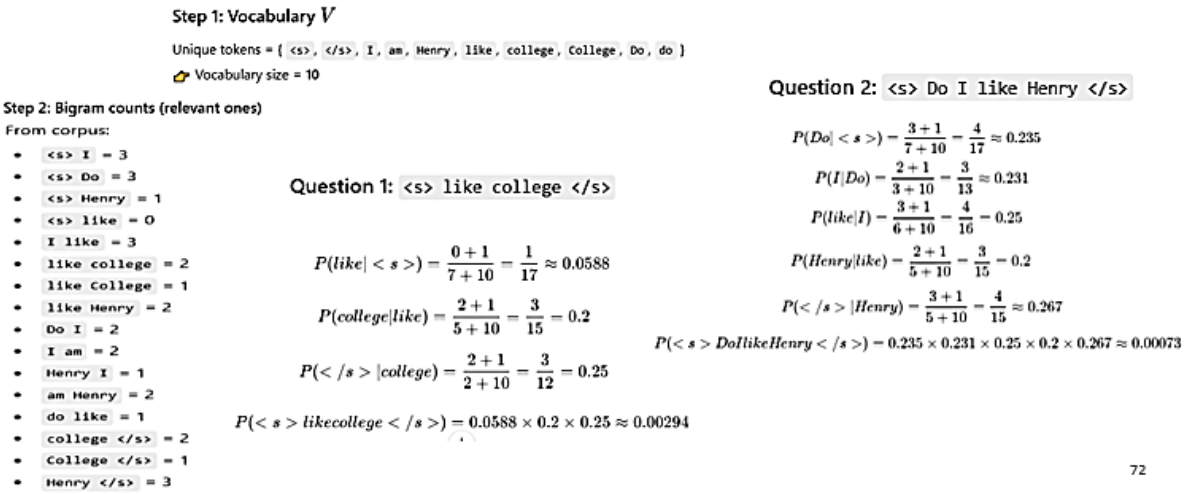
Consider the following training data:

<s> I am Henry </s>
<s> I like college </s>
<s> Do Henry like college </s>
<s> Henry I am </s>
<s> Do I like Henry </s>
<s> Do I like College </s>
<s> I do like Henry </s>

Give the Bigram probability using Laplace Model for:

1. <s> like college </s>

2.<s> Do I like Henry </s>



2. Good-Turing Smoothing

It is a method to re-estimate probabilities of events (words, n-grams) by adjusting their counts.

- Idea: Events that occurred r times should "borrow" some probability mass for unseen events (count = 0).
- Instead of using raw counts, we use adjusted counts: $c^* = (r+1) \cdot \frac{N_{r+1}}{N_r}$

  where: r = observed count of an event, Nr = number of events that occurred exactly r times, c* = adjusted count
- Then the probability is : $P_{GT}(event) = \frac{c^*}{N}$ Where: N is the total number of tokens.

Example-

Suppose vocabulary:

V={cat, dog, lion, tiger, elephant} =3.

Corpus:

cat cat dog dog cat tiger

Counts: Total words N=6

cat = 3, dog = 2, tiger = 1, lion = 0, elephant = 0.

Frequency of frequencies:

N_0=2 (lion, elephant), N_1=1 (tiger), N_2=1 (dog), N_3=1 (cat)

Using Good-Turing formula (adjusted counts):

For r = 0: unseen words $c^* = (0+1) \cdot \frac{N_1}{N_0} = 1 \cdot \frac{1}{2} = 0.5$

For r=1: tiger $c^* = (1+1) \cdot \frac{N_2}{N_1} = 2 \cdot \frac{1}{1} = 2$

For r=2: dog $c^* = (2+1) \cdot \frac{N_3}{N_2} = 3 \cdot \frac{1}{1} = 3$

For r=3: cat $c^* = (3+1) \cdot \frac{N_4}{N_3}$

But N_4=0. Often we leave high counts unchanged or smooth further. So cat ≈ 3 (same).

Total adjusted counts = 0.5+0.5+2+3+3=90.5

Probabilities:

P(cat)=3/9=0.333,
P(dog)=3/9=0.333,
P(tiger)=2/9=0.222
P(lion)=0.5/9=0 .056
P(elephant)=0.5/9=0.056

- *Unlike Laplace (which adds 1 everywhere), Good-Turing uses frequency patterns to redistribute probability.*
- *Works especially well when the vocabulary is large and many words are unseen.*

3. Add-K Smoothing (Lidstone Smoothing)

A simple generalization of Laplace smoothing. Instead of adding 1, you add a small constant *k* (e.g., 0.5, 0.05, 0.01).

Formula:

$$P_{Add\text{-}k}(w_n|w_{n-1}) = Count(w_{n-1}, w_n) + k / Count(w_{n-1}) + k \cdot V$$

Advantage:

The value *k* can be tuned on a development set to find a better balance between seen and unseen events than add-1.

4. Backoff

The idea is to use more detailed n-grams if they are available, but to "back off" to less detailed *(n-1)*-grams when the count is zero.

- Example: To calculate a trigram probability P(world | hello, beautiful), you would:
    1. First check the count of hello beautiful world.
    2. If the count is zero (or too low), you back off to the bigram probability P(world | beautiful).
    3. If that bigram count is also zero, you back off further to the unigram probability P(world).
- A backoff model requires discounting the higher-order n-gram probabilities to free up mass for the lower-order models.

5. Interpolation

Similar to backoff, but instead of choosing one level, you always mix the estimates from all n-gram levels. This is often more effective.

Example (Trigram Interpolation):

- The estimated probability is a weighted sum:

$$P_{interp}(w_n|w_{n-2}, w_{n-1}) = \lambda_1 P(w_n|w_{n-2}, w_{n-1}) + \lambda_2 P(w_n|w_{n-1}) + \lambda_3 P(w_n)$$

where $\lambda_1 + \lambda_2 + \lambda_3 = 1$

- The lambdas (λ) are weights learned on a development set.
- This way, the model always gets information from all levels of context, even if the trigram exists.

6. Kneser-Ney Smoothing

This is one of the most effective and widely used smoothing techniques in practice. It introduces a key insight:

- Standard unigram models assign high probability to common words like "the", "of", "Monday".
- However, if we back off to a unigram, we don't want P(Monday) just because Monday is a common word. We want to know how *likely a word is to appear in an unfamiliar context*

*****