DEPARTMENT OF CSE

23CSX502 - Natural Language Processing

UNIT 2 WORD LEVEL ANALYSIS

MODULE

N-grams Models of Syntax - Counting Words - Unsmoothed N-grams, Evaluating N-grams, Smoothing, Interpolation and Backoff – Word Classes, Part of Speech Tagging, Rule-based, Stochastic and Transformation-based tagging, Issues in PoS tagging – The Viterbi algorithm - Hidden Markov and Maximum Entropy models.

2. N-grams Models of Syntax

An N-gram model is a statistical approach in Natural Language Processing that approximates a language's syntactic structure by analyzing sequences of consecutive words. It is based on the Markov assumption, which states that the probability of a word depends only on the previous N-1 words, rather than the entire sentence history, making computation efficient. In this model, a sentence is tokenized into fixed-length word sequences, such as unigrams (N=1), bigrams (N=2), trigrams (N=3), and so on, to capture local word-ordering patterns that resemble syntax. Using the chain rule, the probability of a sentence $P(w_1, w_2, \dots, w_n)$is decomposed into conditional probabilities, approximated as $P(w_i \mid w_{i-(N-1)}, \dots, w_{i-1})$.

2.1 Language model

- A model that learns to assign probability to word sequences
  Goal → Compute $P(w_1, w_2, \dots, w_n)$.
- The probability of a full sentence is hard to compute directly.
- We approximate using the Markov assumption → a word depends only on the previous (N-1) words.
- This reduces computation and sparsity.

General formula

- $P(w_1^n) \approx \prod_{i=1}^{n} P(w_i \mid w_{i-N+1}^{i-1})$

2.1.1 Counting Words from Corpus

- **Step 1 — Collect Corpus (Example)**

<s> I like NLP </s>
<s> I like ML </s>
<s> I love NLP </s>

- **Step 2 — Tokenization** Break into tokens, including sentence boundary symbols

<s> → start
</s> → end

- **Step 3 — Count Unigrams**

Word	Count
<s>	3
I	3
like	2
love	1
NLP	2
ML	1
</s>	2

- **Step 4 — Count Bigrams -**Pairs of consecutive words

Bigram	Count
<s> I	3
I like	2
I love	1
like NLP	1
like ML	1
love NLP	1
NLP </s>	1
ML </s>	1

- **Step 5 — Count Trigrams -**Triples of consecutive words

Trigram	Count
<s> I like	2
<s> I love	1
I like NLP	1
I like ML	1
I love NLP	1
like NLP </s>	1
like ML </s>	1

2.1.2 Unsmoothed N-gram Probabilities

An N-gram language model is a probabilistic model used to predict the likelihood of a word sequence in a language. It assumes that the probability of a word depends only on the previous $N-1$words. When probabilities are computed directly from observed frequencies without any adjustment, the model is called an unsmoothed N-gram model.

Basic Assumption Markov Property

Unsmoothed N-gram models rely on the Markov assumption, which simplifies language modeling by limiting context

$$P(w_1, w_2, \dots, w_n) \approx \prod_{i=1}^{n} P(w_i \mid w_{i-(N-1)}, \dots, w_{i-1})$$

Examples

- Unigram assumes word independence
- Bigram depends on the previous word
- Trigram depends on the previous two words. This assumption reduces computational complexity but introduces limitations in the model**.**

Bigram Probability

$$P(w_i \mid w_{i-1}) = \frac{count(w_{i-1}, w_i)}{count(w_{i-1})} \text{ (or)} \qquad P(word \mid previous) = \frac{count(previous, word)}{count(previous)}$$

Now substitute counts

Examples: Consider the same corpus

$$P(like \mid I) = \frac{2}{3} = 0.66$$
$$P(love \mid I) = \frac{1}{3} = 0.33$$

Trigram Probability

$$P(w_i \mid w_{i-2}, w_{i-1}) = \frac{count(w_{i-2}, w_{i-1}, w_i)}{count(w_{i-2}, w_{i-1})}$$

Examples

$$P(NLP \mid I, like) = \frac{1}{2} = 0.5$$
$$P(ML \mid I, like) = \frac{1}{2} = 0.5$$
$$P(NLP \mid I, love) = \frac{1}{1} = 1.0$$

Target sentence we want probability for <s> I like NLP </s>

Bigram	How many times it appeared (count of the pair)	Count of previous word (just that first word alone in the whole corpus)	How we got the previous word count (check corpus step)	Probability
<s> I	3	3	<s> appears in all 3 sentences as start → so count(<s>) = 3	3/3 = 1.0
I like	2	3	I appears once in each sentence → 3 total → so count(I) = 3	2/3 = 0.66
like NLP	1	2	like appears in sentence 1 and 2 only → 2 total → so count(like) = 2	1/2 = 0.5
NLP </s>	2	2	NLP appears in sentence 1 and 3 → 2 total → so count(NLP) = 2	2/2 = 1.0

2.1.3 Evaluating N-grams -Evaluating N-gram language models is the process of measuring how well a model predicts word sequences in unseen text. After training the model on a corpus and estimating N-gram probabilities, its performance is tested on a separate dataset to avoid overfitting. The most widely used evaluation metric is perplexity, which reflects how uncertain or "confused" the model is when predicting the next word in a sequence; lower perplexity values indicate a better-performing model.

Formula

$$PP(W) = 2^{-\frac{1}{N}\sum_{i=1}^{N} log_2 P(w_i|context)}$$

- Lower PP = Better model
- Higher PP = Poor model

Take down the same example. <s> I like NLP </s>

Bigram	Bigram Probability	$\log_2$
<s> I	1.0	0
I like	0.66	−0.599
like NLP	0.5	−1
NLP </s>	0.5	−1

Substitute

$$PP = 2^{-\frac{1}{4}(0-0.599-1-1)}$$

$$PP = 2^{-\frac{1}{4}(-2.599)} = 2^{0.6497} \approx 1.57$$

Interpretation

- PP ≈ 1.57 → very low → model predicts this test sentence well.

2.1.4 Smoothing

In N-gram language modeling, a core problem is the occurrence of zero probability for unseen N-grams — sequences that appear in test data but were never observed in the training corpus. Since unsmoothed models use Maximum Likelihood Estimation (MLE), they assign probability purely on observed counts, so any unseen bigram or trigram gets a probability of 0. When computing sentence probability as a product of conditional probabilities, even a single zero term makes the whole sentence probability 0, which is undesirable and unrealistic. Smoothing techniques address this by adding small artificial values called pseudo-counts to redistribute probability mass to rare and unseen word sequences while preserving normalization.

The fundamental mathematical form of smoothing, known as Lidstone smoothing, is

$$P_{smooth}(w \mid h) = \frac{count(h,w) + \alpha}{\sum_{w'} count(h,w') + \alpha V}$$

where

- $count(h,w)$is the number of words w followed the history/context h in training,
- $\alpha$is the smoothing parameter (pseudo count),
- $V$is the vocabulary size (total unique words in training).

A special case, Laplace smoothing, equal to 1, and Jeffreys-Perks smoothing sets alpha equal to. 5, which both prevent zero probabilities but differ in correction strength. To further maintain the proportional contribution of pseudo counts, also describe the effective count adjustment

$$c_i^* = \frac{(c_i + \alpha)M}{M + \alpha V}$$

This ensures the probability mass is fairly redistributed without overpowering real observations.

Example (Bigram case)

Corpus

<s> I like NLP </s>

<s> I like ML </s>

<s> I love NLP </s>

From bigram counts

- count (I, like) = 2
- count (I) = 3
- Vocabulary $V = 5 \rightarrow$ {I, like, love, NLP, ML}

Applying smoothing

$$P_{smooth}(like \mid I) = \frac{2+1}{3+5} = 3/8 = 0.375$$

Without smoothing = 0.66 (significant probability), but unseen cases = 0.

With smoothing in unseen cases, get a small non-zero probability (small probability) model becomes reliable. Thus, smoothing enables the model to assign realistic probabilities to all valid sentences, improves generalization, reduces perplexity, and ensures unseen sequences are not incorrectly treated as impossible.

Example (with and without smoothing)

Understand with an analogy

Imagine corpus sentences all start with **I** (3 times)

I like → 2 times

I love → 1 time

Without smoothing

You only trust what you saw.Out of 3 times "I" appeared, 2 times "like" came → **66% chance.**

$$P(like \mid I) = 2/3 = 0.66$$

But if the test sentence has a new pair **I play**

Model says

$$P(play \mid I) = \frac{0}{3} = 0 \quad (impossible)$$

That is the real problem.

With smoothing (α = 1)

You now add 1 imaginary count to **every possible next word**, even if you never saw it.

So now the model thinks like this

Next word after "I"	Real count	+1 pseudo count
like	2	3
love	1	2
play (never seen)	0	1
hate (never seen)	0	1
eat (never seen)	0	1

Total possible words after I = **V = 5**
So the denominator also increases

$$P_{smooth}(like \mid I) = \frac{3}{3+5} = 3/8 = 0.375$$

The probability of "like" decreased because the model is now sharing probability with new words like play, hate, and eat, rather than assigning 0 probability to them.
Now I play

$$P(play \mid I) = \frac{1}{8} = 0.125 \; small \; but \; possible$$

Without smoothing	With smoothing
Uses only real counts	Uses absolute + pseudo counts
Seen pair 2/3 = 0.66	Seen pair 3/8 = 0.375 (reduced slightly)
Unseen pair 0	Unseen pair small non-zero value
The sentence can become 0	A sentence always gets a probability.
Not reliable	More reliable and generalizable

2.1.5 Interpolation

In N-gram language modeling, interpolation is a method for combining multiple N-gram orders rather than relying on a single fixed *n*. The model estimates the probability of the next word by taking a weighted sum of unigram, bigram, and trigram probabilities. The formula shown in the image is a typical interpolated trigram model

$$P_{interp}(w_m \mid w_{m-1}, w_{m-2}) = \lambda_3 P_3^*(w_m \mid w_{m-1}, w_{m-2}) + \lambda_2 P_2^*(w_m \mid w_{m-1}) + \lambda_1 P_1^*(w_m)$$

Where

- P_n^* is the probability from the **unsmoothed (raw count) N-gram model**,
- λ_n are the **weights assigned** to each model order,
- and all weights must satisfy the constraint

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

This constraint ensures that the result remains a valid probability distribution.

Assign example interpolation weights.

(Just for understanding; in real models these are learned using EM or tuning)

$$\lambda_1 = 0.2, \lambda_2 = 0.3, \lambda_3 = 0.5$$

Check

$$0.2 + 0.3 + 0.5 = 1$$

Compute interpolated probability

$$P_{interp}(NLP \mid I, like) = 0.5 \times 0.5 + 0.3 \times 0.5 + 0.2 \times 0.13$$
$$= 0.25 + 0.15 + 0.04 = 0.426$$

So

$$P_{interp}(NLP \mid I, like) = 0.426$$

2.1.6 Backoff -It is a fallback strategy in N-gram language models to handle unseen higher-order N-grams. The model first tries to use trigram/bigram probabilities learned from training. If the required pair/sequence was never seen (count = 0), it falls back to a lower-order model (bigram → unigram). A backoff weight $\alpha(h)$scales the fallback probability of keeping the final distribution valid. Backoff follows a hierarchical fallback approach — if a higher order N-gram count is zero, the model "backs off" to a lower order probability multiplied by a backoff weight $\alpha(h)$. The backoff probability is defined as

Backoff Formula

$$P_{bo}(w \mid h) = \begin{cases} \dfrac{count(h,w)}{count(h)}, & \text{if } count(h,w) > 0 \\ \alpha(h) \times P_{lower}(w \mid h'), & \text{otherwise} \end{cases}$$

Where

- h= current context/history
- h'= shortened context when backing off
- P_{lower}= lower-order model probability
- $\alpha(h)$= backoff weight (0 < α < 1)

Count unigrams (individual words)- (These are used if we backoff to unigrams)

List learned words including <s>(3)

I (3), like (2), love (1), NLP (2), ML (1), </s> (3)

Total tokens

$$M\ (total\ tokens) = 3 + +2 + 1 + 2 + 1 + 3 = 15$$

We evaluate an unseen bigram -" like Python."

Step 1 — Try bigram MLE

$$P(Python \mid like) = \frac{count(like, Python)}{count(like)}$$

Check training

- like Python was never seen = 0
- like alone appeared 2 times = 2

So

$$P(Python \mid like) = 0/2 = 0$$

Bigram failed → trigger backoff.

Step 2— Compute lower order probability = Unigram

Now, compute the probability of Python using the **smoothed unigram** (because the raw unigram is also 0)

Lidstone unigram smoothing (α=1 example)

$$P_{lower}(Python) = \frac{0+1}{2+7} = 1/9 \approx 0.11$$

(Here 5 is the vocabulary size for content words {I, like, love, NLP, ML)

Step 3 — Apply Backoff with weight

Assume backoff weight for history like

$$\alpha(like) = 0.4$$

Now compute the final backoff probability

$$P_{bo}(Python \mid like) = 0.4 \times 0.11 = 0.04$$

Final understanding in words

Stage	**What happens**
Bigram exists?	No → probability 0
We fall back to	Lower-order unigram
We scale it by	Backoff weight α(like)=0.4
Final probability	0.02 (small but valid, not zero)

2.1.7 Word Classes -Word Classes in NLP refer to grouping words into syntactic or semantic

categories to reduce sparsity and improve the reliability of language models, tagging, and parsing. Instead of computing N-gram probabilities over exact words, the model operates on classes (clusters), if words in the same class behave similarly in context. A popular method described in *Speech and Language Processing* is IBM/Distributional clustering, a form of hard clustering in which each word belongs to only one class. In a class-based N-gram model, sentence probability is approximated by first computing the probability of a class sequence, followed by the probability of the word given its class. Probability = class probability × word inside class. Reduces sparsity, avoids zero probability, and improves generalization

$$P(w_i \mid h) \approx P(c_i \mid c_{i-1}) \times P(w_i \mid c_i)$$

Where

- $c_i = class(w_i)$
- $h = previouswordcontext(history)$
- $P(c_i \mid c_{i-1})$= class bigram probability
- $P(w_i \mid c_i)$= word probability inside its class

Step	What to do	Formula / Rule	Result	Explanation
1	Take the training corpus	—	3 sentences	<s> I like NLP </s> <s> I like ML </s> <s> I love NLP </s>
2	Define Word Classes	Group words by behavior	C1, C2, C3, C4	C1 (Tech) = {NLP, ML} C2 (Preference) = {like, love} C3 = <s> C4 = </s>
3	Convert corpus to class sequence	Replace each word with a class	All 3 lines same structure	C3 I C2 C1 C4C3 I C2 C1 C4C3 I C2 C1 C4
4	Count class bigram pairs	Count (prevClass, nextClass)	C2 C1 = 3	Every preference word (like/love) was followed by a tech word → 3 times total
5	Count the previous class with ANY next class	Count (prevClass, *)	Count (C2, *) = 3	C2 occurred 3 times in the corpus and had *some next class every time* (here always C1)
6	Compute the class bigram probability	$P(c_i \mid c_{i-1}) = \frac{count(c_{i-1},c_i)}{count(c_{i-1},*)}$	$P(C1 \mid C2) = 3/3 = 1.0$	C2 appeared 3 times, and all 3 times followed by C1

Step	What to do	Formula / Rule	Result	Explanation
7	Compute word probability inside its class	$P(w \mid c) = \frac{count(w \in c)}{count(c)}$	$P(NLP \mid C1) = 2/3 = 0.66$	C1 had 3 words total (NLP, NLP, ML) where NLP=2
8	Compute the final class-based probability	$P(w_i \mid h) \approx P(c_i \mid c_{i-1}) \times P(w_i \mid c_i)$	$P(NLP \mid like) \approx 1.0 \times 0.66 = 0.66$	Class gives structure, inner class gives real-world probability

- Word Classes = group words before modeling
- Hard clustering = 1 word → 1 class
- Probability = class probability × word inside class
- Reduces sparsity, avoids zero probability, improves generalization

2.2 Part-of-Speech (PoS) Tagging -Part-of-Speech Tagging is a fundamental NLP task that assigns grammatical labels (tags) such as noun, verb, adjective, pronoun, adverb, etc., to each word in a sentence. The purpose is to identify the syntactic role of words, enabling machines to better understand sentence structure, meaning, and linguistic patterns. PoS tagging is essential for downstream applications, including parsing, named entity recognition, sentiment analysis, machine translation, question answering, speech recognition, and text summarization, as it provides the foundation for linguistic annotation. There are three major paradigms for PoS tagging: Rule-based, Stochastic (probabilistic/statistical), and Transformation-based tagging.

- **Rule-based taggers** rely on manually written linguistic rules and lexicons (dictionaries). They perform deterministic lookup or pattern matching to assign tags. While highly precise in constrained domains, they do not scale well to open text and require heavy expert maintenance.
- **Stochastic taggers**, such as Hidden Markov Models (HMMs), Maximum Entropy Markov Models (MEMMs), Conditional Random Fields (CRFs), and Maximum Entropy (MaxEnt) classifiers, learn tagging behavior from labeled corpora. They estimate tag sequences based on probability rather than fixed rules. These models handle ambiguity by selecting the most likely tag path rather than assigning tags independently.
- **Transformation-based taggers** (e.g., Brill Tagger) begin with an initial tagging (often using unigram or dictionary lookup), then iteratively apply automatically learned correction rules to reduce tagging errors. This hybrid approach balances statistical learning and explainable rule updates.

2.2.1 Rule-based PoS Tagging

Rule-based PoS tagging assigns grammatical labels using predefined linguistic rules and lexicons, without relying on statistical learning. The tagger typically follows a sequence of rule application: dictionary lookup → context rules → morphological rules → disambiguation rules, ensuring deterministic and explainable tagging. Unlike probabilistic models, it does not compute sequence probabilities; instead, it applies rule templates such as *DT* → *JJ* → *NN** for phrase structure and suffix-based rules like -ing → VBG, -ly → RB to tag unknown words. The system's accuracy depends on rule quality and domain coverage, making it precise for structured

text but limited for open-vocabulary language.

Input Sentence

The researcher is learning NLP quickly

Step	Operation	Rule / Formula / Logic	Output after applying the rule
1	Tokenize	Split sentence into tokens	The researcher is learning NLP quickly
2	Dictionary lookup	Use the lexicon if available	The/DT, researcher/NN, is/VBZ, NLP/NNP
3	Morphological rule for unknown word learning	suffix -ing → VBG	learning/VBG
4	Morphological rule for an unknown word quickly	suffix -ly → RB	quickly/RB
5	Context rule check	VBG comes after VBZ, likely the main verb form; RB modifies the previous verb	tags confirmed
6	Consolidate final tags	—	The/DT researcher/NN is/VBZ learning/VBG NLP/NNP quickly/RB

Output

The/DT researcher/NN is/VBZ learning/VBG NLP/NNP quickly/RB

2.2.2 Stochastic PoS Tagging — Theory

Stochastic (statistical) PoS tagging learns tag patterns automatically from a labeled/tagged corpus, using probabilities rather than manual rules. The model's objective is to find the most probable tag sequence T^* for a given sentence W:

$$T^* = argmax_T P(T \mid W)$$

In an HMM (as described in J&M), tagging is modeled using:

- **Transition probability:** $P(t_i \mid t_{i-1})$
- **Emission probability:** $P(w_i \mid t_i)$

The joint probability of words and tags is computed as:

$$P(T, W) = \prod_{i=1}^{n} P(w_i \mid t_i) \times P(t_i \mid t_{i-1})$$

The best tag path is decoded efficiently using the Viterbi dynamic programming algorithm

$$V_t(j) = \max_i [V_{t-1}(i) \times a_{ij}] \times b_j(o_t)$$

Example Corpus: "I like ML"

Stochastic Tagging using HMM (Bigram model)

Component	What we compute	Formula	Count values used	Result
Transition 1	Probability of VBP after PRP	$P(t_i \mid t_{i-1}) = \frac{count(t_{i-1}, t_i)}{count(t_{i-1})}$	Count (PRP, VBP) = 4	$4/6 = 0.66$

Component	What we compute	Formula	Count values used	Result
			count (PRP) = 6	
Transition 2	Probability of NNP after VBP	$P(t_i \mid t_{i-1}) = \frac{count(VBP, NNP)}{count(VBP)}$	Count VBP, NNP) = 3 count (VBP) = 5	$3/5 = 0.6$
Emission 1	Probability that PRP emits I	$P(w \mid t) = \frac{count(word, tag)}{Totalcorpustokens}$	count(I) = 3 Total tokens = 12	$3/12 = 0.25$
Emission 2	Probability that VBP emits like	$P(w \mid t) = \frac{count(word, tag)}{count(tag)}$	count(like, VBP) = 2 count(VBP) = 5	$2/5 = 0.4$
Emission 3	Probability that NNP emits ML	$P(w \mid t) = \frac{count(ML, NNP)}{count(NNP)}$	count(ML, NNP) = 1 count(NNP) = 3	$1/3 \approx 0.33$
Final Tag Sequence	Probability of full tag path for sentence	$P(T, W) = \prod_{i=1}^{n} P(w_i \mid t_i) \times P(t_i \mid t_{i-1})$	— multiply all the above results	$0.25 \times 0.4 \times 0.33 \times 0.66 \times 0.6 \approx 0.0131$

Final Tagged Output "I/PRP like/VBP ML/NNP"

2.2.3 Transformation-based Tagging

Transformation-based tagging is a supervised, rule-learning NLP approach introduced by Eric Brill. It first assigns an initial PoS tag to every word using a simple method (usually a unigram tag or a dictionary). Then, it learns ordered transformation rules from a tagged corpus to correct errors iteratively. The final tag sequence is produced by applying these learned rules in order. This approach is explainable, data-driven, and less sparse than pure N-grams because rules generalize over patterns. The rule application form can be written as:

$$\boldsymbol{tag_{new}(w_i) = apply(rule_k, tag_{current}(w_i), context)}$$

Brill rule template examples:

- If the word is tagged as X and the next word is Y → change X to Z
- If word ends with suffix "ing" → change tag to VBG
- If previous tag is DT and current tag is NN → keep, else transform

Example Training tagged corpus (tiny sample) -I love ML

<s> I/PRP like/VBP NLP/NNP </s>
<s> I/PRP love/VBP ML/NNP </s>

Step	Operation	Rule / Logic	Output
1	Tokenize test sentence	Split into tokens	I love ML
2	Initial tagging	Use dictionary/unigram tag	I/PRP, love/NN, ML/NN *(assume wrong initial tags for demo)*
3	Apply transformation rule 1	Rule learned: Change NN → VBP if previous tag is PRP	love/VBP

Step	Operation	Rule / Logic	Output
4	Apply transformation rule 2	Rule learned: Change NN → NNP if the word is in the known tech lexicon	ML/NNP
5	Check rule order and finalize	No more corrections needed	I/PRP love/VBP ML/NNP

Final tagged sentence
I/PRP love/VBP ML/NNP

Comparison of Rule-based Tagging, Stochastic Tagging, and Transformation-based Tagging (Brill)

Aspect	Rule-based Tagging	Stochastic Tagging (HMM/MaxEnt)	Transformation-based Tagging (Brill)
Nature	Deterministic, hand-crafted rules	Probabilistic, learned from data	Hybrid: learned rules applied sequentially
Needs labeled corpus?	Not required (uses dictionary + grammar rules)	Required	Required (for rule learning)
Uses probability formulas?	No	Yes	No probability at tagging time *(rules applied directly)*
Main formulas	— heuristic rules	(P(w_i	t_i)), (P(t_i
Handles unseen words?	Uses morphology rules (-ing, -ed, numbers, caps)	Gives 0 unless smoothed	Rules generalize, morphology used
Solves ambiguity by	Grammar patterns, priority rules	Best probability path (Viterbi/MaxEnt)	Iterative rule corrections
Example of ambiguity	"light flies" → uses rule patterns	Viterbi chooses the best tag sequence	Brill rules correct initial wrong tags
Explainability	Fully explainable	Less explainable	Fully explainable (rule logs)
Scalability	Hard to maintain for large text	Scales well	Scales well
Typical accuracy	Medium (rule dependent)	High (data dependent)	High (rule + corpus dependent)
Weakness	Manual rules are heavy, domain-limited	Zero probability, label bias risk (MEMM)	Needs good initial tag + rule ordering

2.2.4 Issues in PoS Tagging -Issues in PoS tagging exist because language is ambiguous, sparse, noisy, and evolves beyond training data so we use context, morphology, smoothing, backoff, or better models to fix it.
Part-of-Speech tagging faces several practical and linguistic challenges that affect accuracy, robustness, and generalization of tagging systems. One major issue is lexical ambiguity, in which a single word can take multiple PoS tags depending on context (e.g., "duck" can be a noun or a verb). Another core problem is unknown or Out-of-Vocabulary (OOV) words, which were never seen in the training lexicon or corpus, making it difficult for deterministic or probabilistic taggers to assign reliable tags. Morphologically rich languages introduce complexity because a single root word can generate many surface forms, increasing sparsity in transition/emission counts. Stochastic models like MEMM suffer from the label bias problem due to local normalization, while HMMs assume conditional independence and may not model long-range dependencies well. Domain shift between training and test text (e.g., biomedical vs social media) degrades performance because word usage patterns change. Multi-word expressions (e.g., *"in spite of"*) behave as single functional

units but get split and mistagged. Noisy text from chat, speech transcripts, or ASR errors leads to unreliable tokens. Evaluation of tag sequence in HMM tagging uses Viterbi decoding, but if transition/emission counts are zero, smoothing or backoff must be applied:

$$P(t_i \mid t_{i-1}) = \frac{count(t_{i-1}, t_i) + \alpha}{count(t_{i-1}) + \alpha V}$$
$$P(w_i \mid t_i) = \frac{count(t_i, w_i) + \alpha}{count(t_i) + \alpha V}$$

If a bigram is unseen, backoff uses:

$$P_{bo}(w \mid h) = \alpha(h) \times P_{lower}(w)$$

These issues collectively motivate the use of context-aware tagging, subword modeling, improved smoothing, and discriminative models with global normalization, such as CRFs**,** for reliable PoS tagging in real systems.

Example

Issue Type	**Problem**	**Example**	**How it's solved**
Ambiguity	Word has multiple tags	duck → NN or VB	Use context or probability (Viterbi/MaxEnt/Brill)
OOV words	Word unseen in training	ChatGPTed	Use morphology rules or backoff to lower N-grams
Sparsity	N-gram count = 0	love ML	Backoff or smoothing applied
Label bias	Local normalization error	MEMM bias	Use CRF or global normalization
Domain shift	Training vs test mismatch	Quarry vs. Social Text	Domain adaptation or fine-tuning
Multi-word expr	Split tagging loses meaning	in spite of	Tag as a single class chunk

2.3 Viterbi Algorithm-The Viterbi algorithm is a dynamic programming method to find the most probable sequence of hidden states (PoS tags) $T = t_1^n$ for an observed sentence $W = w_1^n$ using an HMM.
It maximizes:

$$T^* = argmax_T P(T \mid W)$$

The recursion used by Viterbi is:

$$V_i(k) = \max_j [V_{i-1}(j) \times P(t_k \mid t_j)] \times P(w_i \mid t_k)$$

Where

- $V_i(k)$= best probability of tag t_k generating words up to a position i
- $P(t_k \mid t_j)$= transition probability from previous tag t_j to current tag t_k
- $P(w_i \mid t_k)$= emission probability of tag t_k producing word w_i

The algorithm also stores back pointers to reconstruct the best tag path.
Input sentence "I like ML"

Tag Transition	**Count**	**Probability**
PRP → VBP	3	3/3 = 1.0

Tag Transition	Count	Probability
VBP → NNP	2	2/3 = 0.66
VBP → NN	1	1/3 = 0.33

Emission	Count	Tag total	Probability
PRP emits "I."	3	3	3/3 = 1.0
VBP emits "like."	2	3	2/3 = 0.66
NNP emits "ML."	1	2	1/2 = 0.5
NN emits "ML."	1	3	1/3 = 0.33

Viterbi decoding table

Position (i)	Word (w_i)	Current Tag (State)	Compute using	Probability stored in V_i
1	I	PRP	Initial: $P(I \mid PRP) = 1.0$	**1.0**
2	like	VBP	$V_1 \times P(VBP \mid PRP) \times P(like \mid VBP)$ = 1.0 × 1.0 × 0.66	**0.66**
3	ML	NNP	$V_2 \times P(NNP \mid VBP) \times P(ML \mid NNP)$ = 0.66 × 0.66 × 0.5	**0.2178**
3	ML	NN *(alternate path for comparison)*	$V_2 \times P(NN \mid VBP) \times P(ML \mid NN)$ = 0.66 × 0.33 × 0.33	**0.0719**

Step 4 — Choose the best tag for position 3
Compare tag probabilities for ML:

$$NNP = 0.2178, NN = 0.0719$$

Best = **NNP**
Final best tag sequence
I/PRP like/VBP ML/NNP
2.4 Hidden Markov Model (HMM)- HMM is a **generative stochastic model** used for sequence tagging. It assumes that each word depends only on its tag, and that each tag depends only on the previous tag (the Markov assumption). The model selects the best tag sequence by maximizing:

$$T^* = argmax_T P(T \mid W)$$

Joint probability:

$$P(T, W) = \prod_{i=1}^{n} P(w_i \mid t_i) \times P(t_i \mid t_{i-1})$$

Transition:

$$P(t_i \mid t_{i-1}) = \frac{count(t_{i-1}, t_i)}{count(t_{i-1})}$$

Emission:
Vision: To produce demand driven, quality conscious and globally recognized computer professionals through education, innovation and collaborative research.

$$P(w_i \mid t_i) = \frac{count(t_i, w_i)}{count(t_i)}$$

Solved Example (HMM) -Tagged training corpus
<s> I/PRP like/VBP NLP/NNP </s>
<s> I/PRP love/VBP ML/NNP </s>
<s> You/PRP like/VBP ML/NNP </s>
Counts derived

Count	Value
count(PRP, VBP)	3
count(PRP)	3
count(VBP, NNP)	2
count(VBP)	3
count(NNP, ML)	2
count(NNP)	3
VBP emits "love"	1
NNP emits "ML"	1

Compute:

$$P(VBP \mid PRP) = 3/3 = 1.0$$
$$P(ML \mid NNP) = 1/3 \approx 0.33$$
$$P(NNP \mid VBP) = 2/3 \approx 0.66$$

Tagging I love ML:

$$P(T, W) = 1.0 \times 1/3 \times 1/3 \times 1.0 \times 0.66 \approx 0.0131$$

Final tags: I/PRP love/VBP ML/NNP
2.5 Maximum Entropy (MaxEnt) Model
MaxEnt is a discriminative probabilistic classifier, not a generative one. It predicts tag using features (context words, suffixes, capitalization, etc.) and does not assume independence, unlike HMMs. It computes:

$$P(t \mid w) = \frac{e^{(w \cdot f(t,w))}}{\sum_{t'} e^{(w \cdot f(t',w))}}$$

Where:

- $f(t, w)$= feature function
- w= learned weights
- The model chooses the tag with highest probability:

$$t^* = argmax_t P(t \mid w)$$

Solved Example (MaxEnt)
Sentence: "NPTEL students learn ML"
Extract features for the word **learn**:

Feature	Value
Previous word is noun (NPTEL/DT students/NN)	Yes
Word ends with "-n" or "-ing"?	no
Next word is proper noun ML	Yes
Capitalized?	no

Model evaluates using log-linear scoring and predicts the highest-probability tag: learn → VBP (verb)
Tagged output:NPTEL/NNP students/NNS learn/VBP ML/NNP

Compare of HMM and MaxEnt

Aspect	HMM	MaxEnt
Type	Generative	Discriminative
Learns	Transition + Emission	Feature weights
Independence assumption	✓ Yes	❌ No
Handles ambiguity by	Best tag sequence (Viterbi)	Best tag per word using features
Formula nature	Multiplicative	Log-linear softmax
Fails if unseen?	Yes unless smoothed	❌ No, uses feature generalization
Explainable?	Medium	High (feature based)