UNIT 2

CONTENTS

- 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 – Hidden Markov and Maximum Entropy models.

An **N-gram** is a sequence of **N consecutive words** in a text.

Unigram (1-gram): single word (e.g., *"the"*, *"dog"*).

Bigram (2-gram): two consecutive words (e.g., *"the dog"*).

Trigram (3-gram): three consecutive words (e.g., *"the dog barked"*).

N-gram models are used in NLP to estimate the probability of a word based on the previous (N-1) words. This helps in tasks like text prediction and speech recognition.

An **unsmoothed N-gram model** calculates the probability of a word sequence **exactly as it appears in the training data**. It does **not make any adjustments** for word sequences that were not observed during training.

The general formula for an N-gram model is:

$$P(w_n|w_{n-1},\ldots,w_{n-N+1}) = \frac{\text{Count}(w_{n-N+1},\ldots,w_n)}{\text{Count}(w_{n-N+1},\ldots,w_{n-1})}$$

Example for a bigram (2-gram):

$$P(\text{sat}|\text{cat}) = \frac{\text{Count}(\text{cat sat})}{\text{Count}(\text{cat})}$$

Suppose we have the training corpus:
“the cat sat”, *“the dog sat”*, *“the dog barked”*, *“cat barked”*

Unigram counts:

- the = 3, cat = 2, dog = 2, sat = 2, barked = 2

Bigram counts:

- the cat = 1, cat sat = 1, the dog = 2, dog sat = 1, dog barked = 1, cat barked = 1

Now, probability:

$$P(\text{sat}|\text{cat}) = \frac{\text{Count}(\text{cat sat})}{\text{Count}(\text{cat})} = \frac{1}{2} = 0.5$$

In unsmoothed models, **unseen word sequences get a probability of zero**, even if they are valid.

Example:
We never saw the bigram *"cat dog"* in the training corpus.

$$P(\text{dog}|\text{cat}) = \frac{\text{Count}(\text{cat dog})}{\text{Count}(\text{cat})} = \frac{0}{2} = 0$$

This is a big limitation because if even one bigram in a sentence has a zero probability, the entire sentence's probability becomes zero.

Limitations :

Unsmoothed N-grams are simple and work well when the training data is large and complete.

But they **fail when encountering new word combinations**, which makes them unreliable for real-world language tasks.

Because of this, **smoothing techniques** like **Laplace Smoothing**, **Good-Turing**, or **Kneser-Ney** are applied to assign small non-zero probabilities to unseen sequences.

Smoothing

Smoothing is a technique used in N-gram models to adjust probability estimates in order to handle situations where certain word sequences do not appear in the training corpus (resulting in zero probability for those sequences).

It helps ensure that the model assigns non-zero probabilities to previously unseen word sequences, improving generalization and robustness.

To solve the zero-probability issue, **smoothing techniques** are applied. These techniques adjust the probability distribution so that unseen word sequences get small non-zero probabilities, ensuring the model generalizes better to new data.

Types of Smoothing Techniques

Laplace Smoothing (Add-One Smoothing)

This is the simplest technique. We **add 1 to every count**, so that no N-gram has zero probability.

Formula for bigrams:

$$P_{\text{Laplace}}(w_n|w_{n-1}) = \frac{\text{Count}(w_{n-1}, w_n) + 1}{\text{Count}(w_{n-1}) + V}$$

Where:

- **V = Vocabulary size (number of unique words)**

B) Additive (Add-α) Smoothing

Laplace works, but **adding 1 is too big** for large vocabularies (it overestimates unseen N-grams).

Instead, we add a small number **α** (e.g., 0.1 or 0.01):

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

Example:

Using α = 0.1 instead of 1:

$$P(\text{barked}|\text{cat}) = \frac{0 + 0.1}{1 + 0.1 \times 5} = \frac{0.1}{1.5} \approx 0.066$$

Now the adjustment is more balanced.

C) Good-Turing Smoothing

This is a smarter approach:

- It **discounts the probabilities** of frequent N-grams slightly.
- The probability "mass" it saves is **reallocated to unseen N-grams**.

Key idea:
Instead of using raw counts, we use adjusted counts:

$$c^* = \frac{(c+1)N_{c+1}}{N_c}$$

Where:

c = original count of the N-gram

N_c = number of N-grams that appear exactly **c** times

N_{c+1} = number of N-grams that appear **c+1** times

Example:

Suppose "cat barked" (c = 0) is unseen.

If there are 10 N-grams that occurred once (N_1 = 10) and 5 N-grams that occurred zero times (N_0 = 5):

$$c^* = \frac{(0+1)N_1}{N_0} = \frac{1 \times 10}{5} = 2$$

This adjusted count gives a **reasonable probability** even for unseen events.

D) Kneser-Ney Smoothing

- In N-gram models, many possible word combinations (**bigrams, trigrams**) are **never seen** in the training data.
- Simple smoothing methods (like Laplace, Good–Turing) just adjust counts, but they don't handle *contexts* very well.
- Kneser-Ney says: *It's not enough to look at how often a word appears. We must also look at how many different contexts that word appears in.*

Example:

Word **"Francisco"** only follows "San". → even if "Francisco" is frequent, it should not appear after "eat".

Kneser–Ney handles this by focusing on **contexts**, not just raw counts.

Formula (Bigram Model)

$$P_{KN}(w_2|w_1) = \frac{\max(Count(w_1, w_2) - D, 0)}{Count(w_1)} + \lambda(w_1) \cdot P_{KN}(w_2)$$

Where:

- **Count(w_1, w_2)** → number of times bigram (w_1, w_2) appears
- **Count(w_1)** → total times w_1 appears
- **D** → discount value (usually 0.75)
- **$\lambda(w_1)$** → leftover probability weight for unseen cases
- **$P_{kn}(w_2)$** → continuation probability (based on contexts, not frequency)

Suppose we have this training data:

"I like pizza"
"I like pasta"
"You like pizza"
"They eat pizza"

From this, we build counts:

Bigram Counts

- Count("I", "like") = 2
- Count("You", "like") = 1
- Count("They", "eat") = 1
- Count("like", "pizza") = 1
- Count("like", "pasta") = 1
- Count("eat", "pizza") = 1

Word Counts

- Count("I") = 2
- Count("You") = 1
- Count("They") = 1
- Count("like") = 3
- Count("eat") = 1

Kneser–Ney for P(pizza | like)

Step 1: Discount Seen Counts

$$\frac{\max(Count(like, pizza) - D, 0)}{Count(like)}$$

- Count("like", "pizza") = 1
- Count("like") = 3
- D = 0.75

$$= \frac{\max(1 - 0.75, 0)}{3} = \frac{0.25}{3} \approx 0.083$$

Step 2: Compute Back-off Weight λ(like)

$$\lambda(like) = \frac{D \cdot (\text{Number of unique continuations after "like"})}{Count(like)}$$

After "like", we saw {"pizza", "pasta"} → 2 unique continuations

- Count("like") = 3

$$\lambda(like) = \frac{0.75 \cdot 2}{3} = 0.5$$

Step 3: Compute Continuation Probability P_{kn}(pizza)

$$P_{KN}(pizza) = \frac{\text{Number of unique contexts where "pizza" appears}}{\text{Total unique bigram types}}$$

"pizza" appears in 2 contexts: {like pizza, eat pizza}

Total unique bigrams = 6

$$P_{KN}(pizza) = \frac{2}{6} = 0.333$$

Step 4: Final Probability

$$P_{KN}(pizza|like) = 0.083 + 0.5 \cdot 0.333$$

$$= 0.083 + 0.167 = 0.25$$

So, $\mathbf{P_{kn}(pizza \mid like) = 0.25}$

After building an N-gram model (like unigram, bigram, or trigram), we need to test **how good or bad it is** at predicting or generating words. This is called **evaluation**. It helps us know if the model will work well when given new sentences. A good model should be able to predict the next word correctly and also handle different kinds of text smoothly. Without evaluation, we won't know how useful the model really is.

Perplexity tells us **how uncertain or "confused" the model is** when predicting words.

If a model assigns **high probabilities** to the correct words → **low perplexity = good model**.

If it assigns **low probabilities → high perplexity = bad model**.

Formula:

$$\text{Perplexity}(P) = 2^{H(P)} \quad \text{where} \quad H(P) = -\frac{1}{N}\sum_{i=1}^{N} \log_2 P(w_i \mid w_{i-1}, \ldots, w_{i-n+1})$$

- *N*: number of words in the sentence
- *P*(wi|wi−1,...,wi−n+1): predicted probability of word *wi* given the previous *n*−1 words.

Example:

For the sentence:
“I want to eat pizza”, suppose the trigram model gives these probabilities:

- P(want | <s>, I) = 0.2
- P(to | I, want) = 0.5
- P(eat | want, to) = 0.4
- P(pizza | to, eat) = 0.3

$$\text{Perplexity} = \left(\frac{1}{0.2 \times 0.5 \times 0.4 \times 0.3}\right)^{1/4} = \left(\frac{1}{0.012}\right)^{1/4} \approx 4.24$$

Lower perplexity means better performance.

2. Accuracy

Accuracy measures **how many times the model predicted the exact correct next word**.

Formula:

$$\text{Accuracy} = \frac{\text{Correct predictions}}{\text{Total predictions}} \times 100$$

Example:

Suppose on a test set of 10 sequences, the model got the next word right 7 times:

$$\text{Accuracy} = \frac{7}{10} \times 100 = 70\%$$

3. Cross-Validation

When the dataset is small, we split it into parts (folds) and test the model **multiple times**. This avoids bias and gives a more reliable score.

- **Process:**

- Divide dataset into 5 folds (say).
- Train on 4 folds, test on 1.
- Repeat 5 times (each fold gets used once as test set).
- Take the **average accuracy or perplexity**.

4. Log-Likelihood

It measures **how much the model "likes" the given data**, i.e., how probable it thinks the sentence is.

Formula:

$$\text{Log-Likelihood} = \sum_{i=1}^{N} \log P(w_i \mid w_{i-1}, \ldots, w_{i-n+1})$$

Example:

If probabilities for each word are:

- 0.5, 0.4, 0.2

Then:

$$\log(0.5)+\log(0.4)+\log(0.2)=-0.301-0.398-0.699=-1.398$$

Less negative (i.e., higher) log-likelihood = better model.

5. Precision, Recall, F1-Score

These are usually used when you're comparing **model output to reference output** (like in translation or spelling correction).

Formula:

$$\text{Precision} = \frac{\text{Correct outputs predicted}}{\text{Total outputs generated}}$$

$$\text{Recall} = \frac{\text{Correct outputs predicted}}{\text{Total correct outputs in reference}}$$

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Example:

- Model generated: 10 words
- 6 of them match reference (Precision = 6/10 = 60%)
- Reference had 8 correct words (Recall = 6/8 = 75%)

Then:

$$F1 = 2 \cdot \frac{0.6 \cdot 0.75}{0.6 + 0.75} = 66.6\%$$

6. BLEU Score

Checks how similar a machine-generated sentence is to a reference sentence.

- **Example:**

Reference: “He ate the pizza”
Generated: “He eats the pizza”

Compare unigrams (1-grams):

- Common: “He”, “the”, “pizza” = 3/4
→ Precision = 75%

Bigram overlap:

- Reference: “He ate”, “ate the”, “the pizza”
- Generated: “He eats”, “eats the”, “the pizza”
  Only “the pizza” matches = 1/3 → 33.3%

BLEU = geometric mean of unigram and bigram precision
(with brevity penalty if sentence is shorter)

7. Test Set Performance

Meaning: Measures how well the model performs on unseen data.

Example:

Train on 80% of your corpus, test on 20% unseen data.
On test set:

Accuracy = 70%

Perplexity = 3.2

8. OOV Rate (Out-of-Vocabulary)

Meaning: Percentage of test words the model has never seen before.

Example:

Test sentence: “I adore spaghetti”

Training set only had: “I love pasta”

OOV word = “adore”, “spaghetti” → 2 words
Total words = 3

$$\text{OOV rate} = \frac{2}{3} \times 100 \approx 66.7\%$$

Back-off Smoothing

Back-off smoothing is used in **N-gram models (like bigrams, trigrams)** when the exact N-gram we want is **not found** in the training data.

- If the **trigram** (3-word sequence) probability is zero → we **back off** and use the **bigram** (2-word sequence).
- If the bigram probability is also zero → we **back off** to the **unigram** (single word frequency).

This way, we **never get zero probability**, even for unseen sequences.

Without Back-off

We want the probability of predicting the word **"pizza"** after the phrase **"I want to eat"**.

$P(\text{pizza}|\text{I want to eat})$

If "I want to eat pizza" **never appeared in the training data**, an **unsmoothed trigram model** would assign:

$P(\text{pizza}|\text{I want to eat})=0$

This is unrealistic because "pizza" is a valid continuation.

Back-off Step by Step

Step 1: Check trigram probability

We first check:

P(pizza|want to eat) If it's not seen in training data → probability = 0 → **back off to bigram**.

Step 2: Check bigram probability

Now we check:

- P(pizza|eat)
- If "eat pizza" appeared in training data, we can calculate its probability:

$$P(\text{pizza}|\text{eat}) = \frac{\text{Count}(\text{eat pizza})}{\text{Count}(\text{eat})}$$

- If it's not seen, we **back off further** to unigram probability.

Step 3: Check unigram probability

If "eat pizza" bigram is also unseen:

$$P(\text{pizza}) = \frac{\text{Count}(\text{pizza})}{\text{Total words}}$$

This gives a non-zero probability because "pizza" has likely appeared somewhere else in the corpus (e.g., "pizza is tasty", "order pizza").

Interpolation

- **Interpolation** is a smoothing technique used in language modeling to improve the reliability of probability estimates.
- Instead of depending only on the higher-order N-gram (like trigrams), interpolation combines probabilities from different N-gram levels such as trigrams, bigrams, and unigrams.
- This is important because higher-order N-grams are more accurate but suffer from data sparsity (many word combinations do not appear in the training corpus).
- Lower-order N-grams are more frequent and robust, though less specific. By interpolating them together, the model balances accuracy and robustness.

Formula

The interpolated probability of a word w_n given its context is calculated as:

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

Here, the first term is the trigram probability, the second term is the bigram probability, and the third is the unigram probability.

The weights ($\lambda 3$,$\lambda 2$,$\lambda 1$) are chosen so that they add up to 1. These weights decide how much importance is given to each N-gram level.

In practice, these weights are tuned using a validation dataset to maximize prediction accuracy.

Example

Suppose we want to estimate the probability of the word "sat" in the context of the phrase "the cat". We have the following probabilities from our model:

- P(sat | the, cat)=0.2 (trigram)
- P(sat | cat)=0.1 (bigram)
- P(sat)=0.05 (unigram)

Now, assume the interpolation weights are chosen as:

$\lambda 3=0.5, \quad \lambda 2=0.3, \quad \lambda 1=0.2.$

The interpolated probability is:

- P_{interp} (sat | the, cat)=(0.5×0.2)+(0.3×0.1)+(0.2×0.05) = 0.1 + 0.03 + 0.01 = 0.14.

Thus, instead of relying only on the trigram, interpolation combines all levels and produces a more reliable probability of **0.14** for the word "sat" following "the cat".

Word classes

- Word classes, also known as parts of speech, are the **basic categories into which words are grouped** based on their function in a sentence.
- These classes help us understand the **structure and meaning** of language.
- Every word we use belongs to one of these classes depending on how it behaves in a sentence.
- For example, some words may name things, some show actions, while others describe or link different words together.

1. Nouns (N):

Words that **name** a **person**, **place**, **thing**, or **idea**.
Examples:

- *dog* (thing), *city* (place), *happiness* (idea)

2. Pronouns (PRON):

Used to **replace nouns** to avoid repetition.
Examples:

- *he*, *she*, *it*, *they*

3. Verbs (V):

Show **actions**, **states**, or **occurrences**.
Examples:

- *run*, *eat*, *exist*, *is*

4. Adjectives (ADJ):

Describe or modify **nouns**, giving **more detail**.
Examples:

- *beautiful*, *green*, *tall*

5. Adverbs (ADV):

Modify **verbs**, **adjectives**, or **other adverbs**. They often tell us **how**, **when**, **where**, or **how much**.
Examples:

quickly, *very*, *here*

6. Prepositions (PREP):

Show **relationships** especially about **location**, **direction**, or **time**.
Examples:

- *in*, *on*, *under*, *between*

7. Conjunctions (CONJ):

Connect **words**, **phrases**, or **clauses**.
Examples:

- *and*, *but*, *because*

8. Interjections (INTJ):

Short words that show **emotion** or **reactions**.
Examples:

- *wow*, *ouch*, *hey*

9. Determiners (DET):

Come **before nouns** and help to specify which noun we're talking about.
Examples:

- *the*, *a*, *some*, *this*

Types of Word Classes (Subcategories)

Some word classes are **further divided** into smaller types.

Nouns

- **Common Nouns**: general names (*city*, *teacher*)
- **Proper Nouns**: specific names (*Paris*, *John*)
- **Countable Nouns**: can be counted (*apple*, *pen*)
- **Uncountable Nouns**: can't be counted (*water*, *sand*)

Types of Word Classes (Subcategories)

Verbs

- **Action Verbs**: show actions (write, run)
- **Stative Verbs**: show a state or condition (belong, exist)
- **Transitive Verbs**: need a direct object
  Example: She eats an apple.
- **Intransitive Verbs**: don't need an object
  Example: He sleeps.
- **Modal Verbs**: show ability, possibility, permission
  Examples: can, could, must, will

Adjectives

- **Comparative**: compares 2 things (*taller*, *smarter*)
- **Superlative**: compares more than 2 things (*tallest*, *smartest*)

Sentence: *The tall boy runs quickly.*

Word	**Word Class**
The	Determiner
tall	Adjective
boy	Noun
runs	Verb
quickly	Adverb

POS Tagging

Part-of-Speech tagging is a **natural language processing (NLP) technique** that involves identifying the word class or grammatical role of each word in a sentence.

It assigns a **label or "tag"** to each word, indicating whether it's a noun, verb, adjective, and so on.

This is not always easy, because many English words can belong to more than one class depending on how they are used.

For example, in *"He can run fast,"* the word *"run"* is a verb, but in *"He went for a run,"* it is a noun.

- Sentence: *I will read the book.*

Word	Tag (Word Class)
I	Pronoun (PRON)
will	Modal Verb (MD)
read	Verb (VB)
the	Determiner (DET)
book	Noun (NN)

Tagged sentence:

I/PRON will/MD read/VB the/DET book/NN

Steps in pos tagging

1. Tokenization

Break the sentence into **words** (tokens).

Example: *The dog runs.*

Tokens: [The, dog, runs]

2. Assign Word Classes

Give the **correct tag** to each word.

Example:

- The → Determiner (DET)
- dog → Noun (NN)
- runs → Verb (VBZ)

3. Check Context (Meaning)

Some words have **more than one meaning**. So we check the sentence to choose the correct tag.

Example:

*I will **book** a ticket.* → **book** is a **verb** (to reserve)

*I read a **book** every day.* → **book** is a **noun** (a thing to read)

So the tag depends on **how the word is used**.

Tag	Word Class	Example
NN	Noun (Singular)	dog
NNS	Noun (Plural)	dogs
PRP	Pronoun	I, he
VB	Verb (base form)	eat, go
VBZ	Verb (3rd person)	eats, runs
VBD	Verb (past)	went, ate
JJ	Adjective	big, happy
RB	Adverb	quickly
DT	Determiner	the, a
IN	Preposition	in, on
CC	Conjunction	and, but
MD	Modal Verb	will, can
UH	Interjection	wow, oh

Sentence: *Wow! She quickly opened the big box.*

Word	**Tag**	**Word Class**
Wow	UH	Interjection
She	PRP	Pronoun
quickly	RB	Adverb
opened	VBD	Verb (Past)
the	DT	Determiner
big	JJ	Adjective
box	NN	Noun

Sentence: The cat sat behind the box.

Word	**Class**
The	
cat	
sat	
behind	
the	
box	

***Sentence**: The cat sat behind the box.*

Word	**Class**
The	Determiner
cat	
sat	
behind	
the	
box	

***Sentence**: The cat sat behind the box.*

Word	**Class**
The	Determiner
cat	Noun
sat	
behind	
the	
box	

***Sentence**: The cat sat behind the box.*

Word	**Class**
The	Determiner
cat	Noun
sat	Verb
behind	
the	
box	

***Sentence**: The cat sat behind the box.*

Word	**Class**
The	Determiner
cat	Noun
sat	Verb
behind	Preposition
the	
box	

***Sentence**: The cat sat behind the box.*

Word	**Class**
The	Determiner
cat	Noun
sat	Verb
behind	Preposition
the	Determiner
box	

***Sentence**: The cat sat behind the box.*

Word	Tag	Class
The		Determiner
cat		Noun
sat		Verb
behind		Preposition
the		Determiner
box		Noun

Rule-Based POS Tagging

Rule-based POS tagging is a traditional method of tagging words in a sentence with their grammatical categories (like noun, verb, adjective, etc.) using a set of **manually written rules**.

It doesn’t rely on machine learning or statistical methods. Instead, it uses **grammar rules** and **a dictionary (lexicon)** to figure out the correct tag for each word based on its form and position in a sentence.

The rule-based approach follows **three main steps**:

1.Lexicon Lookup:

Every word in the sentence is first checked in a **dictionary** that lists possible tags.

For example:

1. "run" → can be both a noun (NN) and a verb (VB)
2. "light" → can be an adjective (JJ) or a noun (NN)

- Linguists or NLP developers take a **huge corpus of real-world text** (news articles, books, etc.) where every word is already tagged with the correct part of speech.
- From this annotated text, they calculate **how often each word appears with each tag**.
- A lexicon (lookup table) is created with each word and its possible tags.
- The tag that appears **most frequently** with a word becomes its **default tag**.

2. Initial Tagging:
The tagger initially assigns the **most common tag** to the word based on dictionary data.
For example:

"run" → tagged as VB (verb), if that's the more frequent use.

Word	Possible Tags	Frequencies	Most Frequent Tag
run	VB (verb), NN (noun)	VB: 700, NN: 300	**VB**
light	NN, JJ	NN: 250, JJ: 450	**JJ**
book	NN, VB	NN: 800, VB: 100	**NN**

3. Applying POS rules:

These rules are called **transformation rules** – they are applied **after** the initial tagging (which is usually based on the most frequent tag) to **correct** the mistakes using **contextual clues**.

Types of Common Rules in Rule-Based PoS Tagging

1. Morphological Rules (Based on word structure)
2. Contextual Rules (Based on nearby words or tags)
3. Default/Fallback Rules

1. Morphological Rules (Based on word structure)

These rules look at the **form or ending** of the word (prefixes, suffixes, etc.).

Rule	Description	Example
If a word ends in -ly, tag it as RB (Adverb)	Common for adverbs	**quickly** → RB
If a word ends in -ing, tag it as VBG (Gerund/Verb)	Common for present participle	**running** → VBG
If a word ends in -ed, tag it as VBD (Past Tense Verb)	Regular past tense verbs	**walked** → VBD
If a word ends in -ness, tag it as NN (Noun)	Nominalization	**happiness** → NN
If a word ends in -ous, tag it as JJ (Adjective)	Adjective rule	**famous** → JJ

2. Contextual Rules (Based on nearby words or tags)

These use the **surrounding words or their tags** to reassign the correct tag.

Rule	Description	Example
If a word is tagged as VB and follows "to", retag it as VB (Infinitive verb)	After “to”, verb likely	"to **eat**" → VB
If a word is tagged as NN and follows "the", retag it as NN	Articles usually followed by nouns	"the **ball**" → NN
If a word is tagged as NN and comes after a JJ (Adjective), keep NN	Adjective + Noun pattern	"red **car**" → NN
If a word is tagged as VB but is preceded by PRP (pronoun), keep VB	PRP + verb is common	"She **runs**" → VB
If a word is tagged as NN and ends in -ly, retag as RB	Fix misclassified adverb	"badly" was NN → RB

3. Default/Fallback Rules

If a word is **unknown** (not in the dictionary), assign a tag based on suffix or context.

Rule	Example
Unknown word ends in -s → Tag as NNS (Plural Noun)	"apples" → NNS
Unknown word ends in -tion → Tag as NN (Noun)	"creation" → NN
Unknown word starts with capital letter → Tag as NNP (Proper Noun)	"India" → NNP

Brill Tagger (Special Rule-based Algorithm)

Brill's Tagger is a **smarter version of rule-based tagging**, created by **Eric Brill**.

It is also based on rules **but** the difference is the rules are **not written manually,** they are **learned automatically from data**.

1.Initial Tagging:
Each word is tagged with its **most common POS tag** (from dictionary or training data).
Example: “runs” is usually a **VBZ** (verb), so it is tagged as such.

2.Error Checking:
It compares the tagging with the **correct tags** from a training corpus.

3.Transformation Rules:
It then finds patterns where the tags were wrong and **learns rules to correct them**.

4.Applies Rules:
These rules are then applied one by one to **fix mistakes**.

Sentence: **“The light bag was heavy.”**

Initial tagging might be:

- "The" → DT
- "light" → **NN** (noun – mistake)
- "bag" → NN
- "was" → VBD
- "heavy" → JJ

Now it finds that "light" is **wrongly tagged** as NN.
Brill’s Tagger learns this pattern:

If a word is tagged as NN but comes **after “The” and before another NN**, change it to **JJ (adjective)**.

So next time it sees: **“The light box”**, it will tag “light” as **JJ**, not NN.