Natural Language Processing
Course code: CSE3015

Module 2
Parts of Speech Tagging

Prepared by

Dr. Venkata Rami Reddy Ch

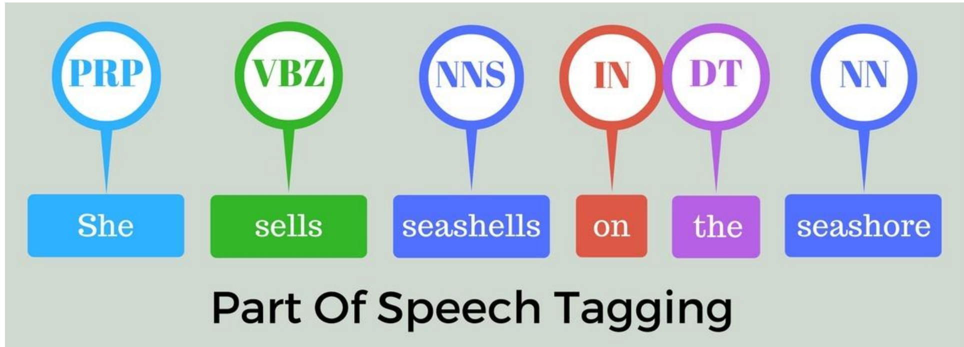
SCOPE

Syllabus

- Parts of Speech Tagging and Named Entities
    - Tagging in NLP,
    - Sequential tagger,
    - N-gram tagger,
    - Regex tagger,
    - Brill tagger,
    - NER tagger;
- Machine learning taggers-MEC, HMM, CRF,

Part-of-Speech (POS) Tagging

- Part-of-speech (POS) tagging is a process in NLP where each word in a text is labeled with its corresponding part of speech.
- Assigning a part-of-speech to each word in a text.
- This can include nouns, verbs, adjectives, and other grammatical categories.
- It helps algorithms understand the grammatical structure and meaning of a text.
- POS tagging is useful for a variety of NLP tasks, such as information extraction, named entity recognition, and machine translation.



Part-of-Speech (POS) Tags

1. **Noun**: A noun is the name of a person, place, thing, or idea.
2. **Pronoun**: A pronoun is a word used in place of a noun.
3. **Verb**: A verb expresses action or being.
4. **Adjective**: An adjective modifies or describes a noun or pronoun.
5. **Adverb**: An adverb modifies or describes a verb, an adjective, or another adverb.
6. **Preposition**: A preposition is a word placed before a noun or pronoun to form a phrase modifying another word in the sentence.
7. **Conjunction**: A conjunction joins words, phrases, or clauses.
8. **Interjection**: An interjection is a word used to express emotion.
9. **Determiner or Article**: A grammatical marker of definiteness (the) or indefiniteness (a, an).

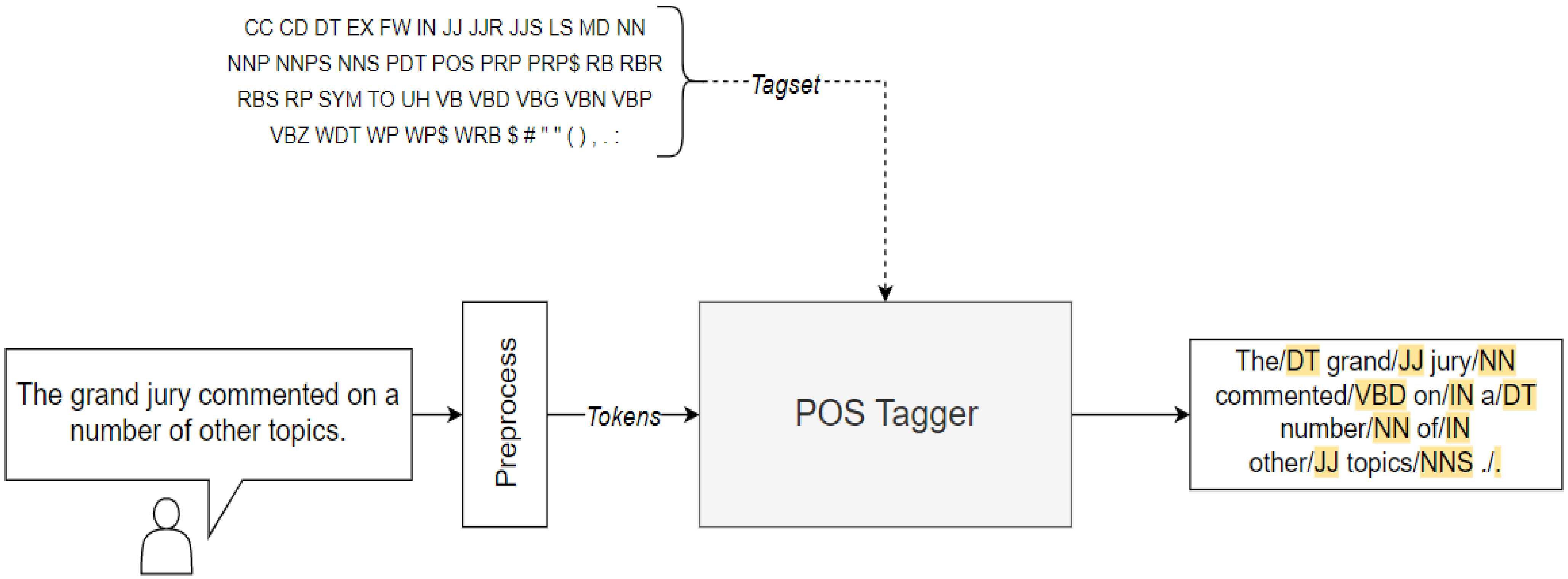
Let’s take an example,

Text: “The cat sat on the mat.”

POS tags:

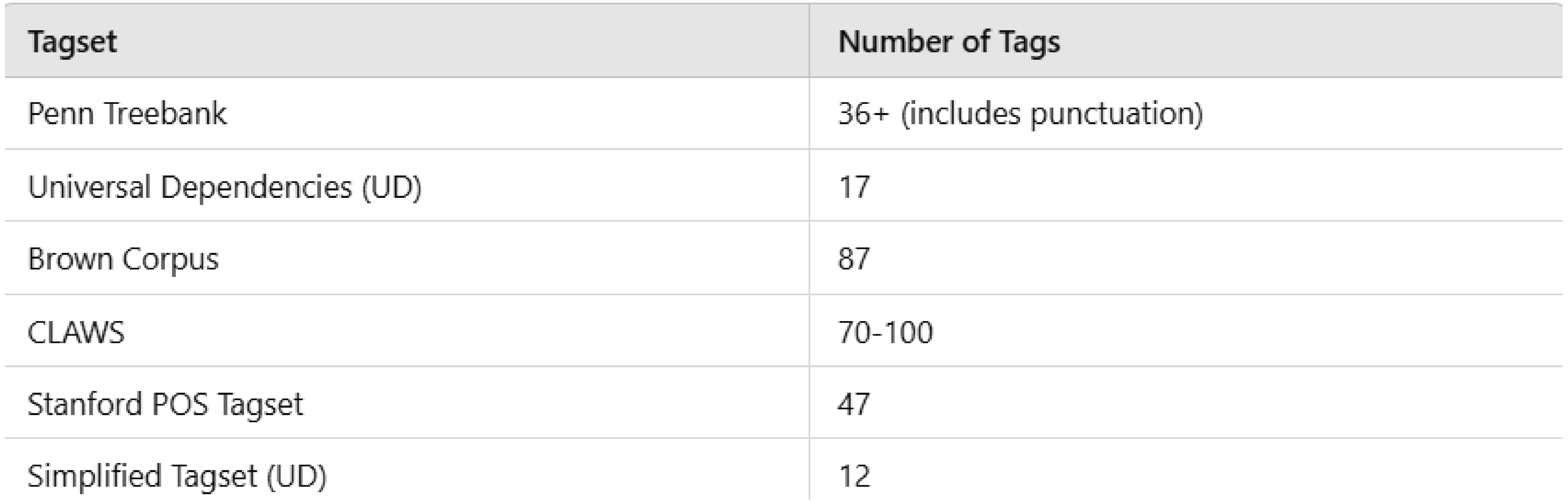
- The: determiner
- cat: noun
- sat: verb
- on: preposition
- the: determiner
- mat: noun

POS Tagging architecture


CC CD DT EX FW IN JJ JJR JJS LS MD NN
NNP NNPS NNS PDT POS PRP PRP$ RB RBR
RBS RP SYM TO UH VB VBD VBG VBN VBP
VBZ WDT WP WP$ WRB $ # " " ( ) , . :
Tagset
The grand jury commented on a number of other topics.
Preprocess
Tokens
POS Tagger
The/DT grand/JJ jury/NN commented/VBD on/IN a/DT number/NN of/IN other/JJ topics/NNS ./.

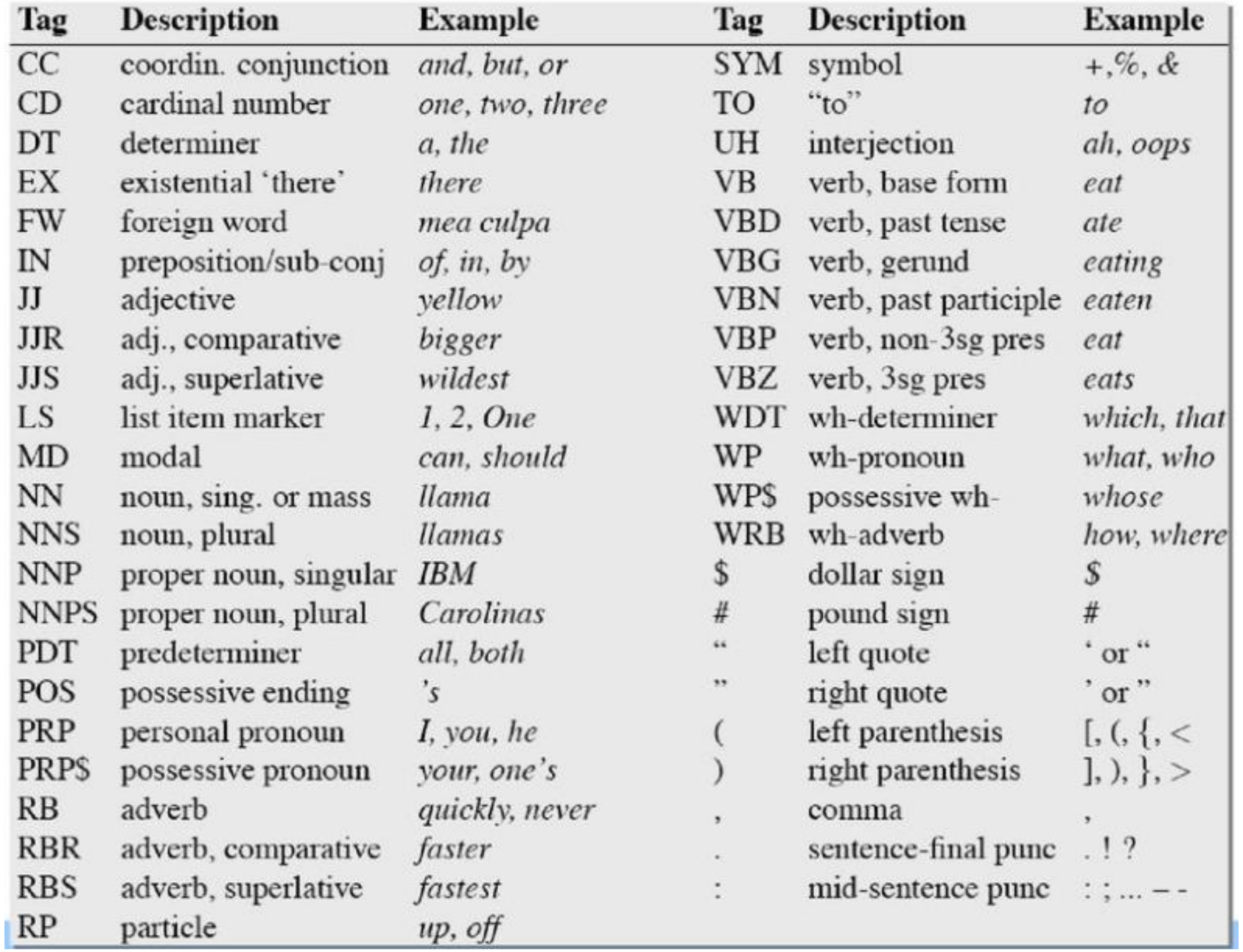
POS Tagset

- Tagset is the collection of tags from which the tagger finds appropriate tags and attaches to the word
- Different POS tag sets are used depending on the language and the application.



Tagset	Number of Tags
Penn Treebank	36+ (includes punctuation)
Universal Dependencies (UD)	17
Brown Corpus	87
CLAWS	70-100
Stanford POS Tagset	47
Simplified Tagset (UD)	12

Penn Treebank POS Tagset



Tag	Description	Example	Tag	Description	Example
CC	coordin. conjunction	*and, but, or*	SYM	symbol	+,%, &
CD	cardinal number	*one, two, three*	TO	“to”	*to*
DT	determiner	*a, the*	UH	interjection	*ah, oops*
EX	existential ‘there’	*there*	VB	verb, base form	*eat*
FW	foreign word	*mea culpa*	VBD	verb, past tense	*ate*
IN	preposition/sub-conj	*of, in, by*	VBG	verb, gerund	*eating*
JJ	adjective	*yellow*	VBN	verb, past participle	*eaten*
JJR	adj., comparative	*bigger*	VBP	verb, non-3sg pres	*eat*
JJS	adj., superlative	*wildest*	VBZ	verb, 3sg pres	*eats*
LS	list item marker	*1, 2, One*	WDT	wh-determiner	*which, that*
MD	modal	*can, should*	WP	wh-pronoun	*what, who*
NN	noun, sing. or mass	*llama*	WP$	possessive wh-	*whose*
NNS	noun, plural	*llamas*	WRB	wh-adverb	*how, where*
NNP	proper noun, singular	*IBM*	$	dollar sign	$
NNPS	proper noun, plural	*Carolinas*	#	pound sign	#
PDT	predeterminer	*all, both*	“	left quote	‘ or “
POS	possessive ending	*’s*	”	right quote	’ or ”
PRP	personal pronoun	*I, you, he*	(	left parenthesis	[, (, {, <
PRP$	possessive pronoun	*your, one’s*	)	right parenthesis	],), }, >
RB	adverb	*quickly, never*	,	comma	,
RBR	adverb, comparative	*faster*	.	sentence-final punc	. ! ?
RBS	adverb, superlative	*fastest*	:	mid-sentence punc	: ; ... – -
RP	particle	*up, off*			

lexicons

- **lexicons** are structured collections of words or phrases that include additional information, such as part of speech, meanings, synonyms, or domain-specific attributes.
- A dictionary that contains words and their possible tags.
- The **WordNet Lexicon** is a widely used lexical database in NLP that groups words into **nouns**, **verbs**, **adjectives**, and **adverbs**.

- For example, the word "run" might have tags such as *verb* or *noun*.

POS Tagger

- A part-of-speech tagger, or **POS-tagger**, processes a sequence of words, and attaches a part of speech tag to each word.
- It is a program that carries out POS Tagging
- Taggers utilize a various types of data: lexicons, dictionaries, rules, etc. for POS tagging.

```
import nltk

nltk.download('averaged_perceptron_tagger')

text = "The quick brown fox jumps over the lazy dog."
tokens = nltk.word_tokenize(text)
# Perform POS tagging
pos_tags = nltk.pos_tag(tokens)

print("Tokenized Words with POS Tags:")
for word, tag in pos_tags:
    print(f"{word}: {tag}")
```

```
Tokenized Words with POS Tags:
The: DT
quick: JJ
brown: NN
fox: NN
jumps: VBZ
over: IN
the: DT
lazy: JJ
dog: NN
.: .
```

- The **Averaged Perceptron Tagger** is the default tagger used in pos_tag().
- **Averaged Perceptron Tagger** assigns tags based on learned features from a large annotated corpus, such as the **Penn Treebank**

Approaches to POS Tagging

- **Rule-based Approach**
  - Uses set of rules to tag input sentences

    e.g. RegExp Tagger
- **Statistical approaches(Machine Learning Based)**
  - Use training corpus to assign a tag to every token in given text.

    e.g N-gram tagger, HMM(Hidden Markov Model), CRF(conditional random field)
- **Transformation based(Hybrid)**

  Rules + machine learning(1 gram tagger)

  e.g Brill Tagger

Regex tagger

- The Regex tagger assigns tags to words based on matching patterns specified using regular expressions.
- You can specify both a regular expression and an associated tag for identifying pos of each word and assign associated tag to that word.
- For instance, we might guess that any word ending in *ed* is the past participle of a verb, and any word ending with *'s* is a possessive noun. We can express these as a list of regular expressions

Determiners:	\b(the|a|an)\b
Adjectives ending in 'able' :	\b\w+able\b'
Past tense verbs:	.*ed$
Adverbs:	.*ly$'
Pronouns:	\b(I|my|he|him|his|she|her|we|you|it|they)\b
Prepositions:	\b(on|in|at|by|with|about|into|to)\b

\b: Word boundary to ensure you're matching whole words.

Write a Python script to tag parts of speech in the given sentence. Define patterns for pronouns, conjunctions, prepositions, determiners, adjectives, verbs, adverbs, and nouns. Then, print the tagged words.

```
import nltk
from nltk.tag import RegexpTagger

# Define patterns for the regular expression tagger
patterns = [
    (r'^\d+$', 'CD'),          # Cardinal numbers
    (r'\b(I|me|my|he|him|his|she|her|we|you|it|they)\b', 'PRP'),  #  pronouns
    (r'\b(on|in|at|by|with|about|into|to)\b', 'IN'),
    (r'\b(and|or|but|also)\b', 'CC'),
    (r'\b(The|the|A|a|An|an)\b', 'DT'),# Determiners
    (r'\b\w+able\b', 'JJ'),     # Adjectives ending in 'able'
    (r'.*ing$', 'VBG'),        # Gerunds
    (r'.*ed$', 'VBD'),         # Past tense verbs
    (r'.*es$', 'VBZ'),         # 3rd person singular verbs
    (r'.*ly$', 'RB'),          # Adverbs
    (r'.*', 'NN')              # Default: Noun
]

# Create a RegexpTagger using the defined patterns
regexp_tagger = RegexpTagger(patterns)
sentence = "John is  running quickly and he catch the 9am train."
words = nltk.word_tokenize(sentence)

# Tag the words using the regular expression tagger
tagged_words = regexp_tagger.tag(words)
print(tagged_words)
```

```
[('John', 'NN'), ('is', 'NN'), ('running', 'VBG'),
('quickly', 'RB'), ('and', 'CC'), ('he', 'PRP'),
('catch', 'NN'), ('the', 'DT'), ('9am', 'NN'), ('train'
'NN'), ('.', 'NN')]
```

Unigram Tagger

- A **Unigram Tagger** is a type of POS tagger that assigns tags based on individual words.
- In a unigram model, the tag for a word is determined independently, without considering the POS tags of the previous words.

How it Works:

Training:

- The Unigram Tagger is trained on a collection of tagged words (word-tag pairs).
- e.g. the Brown Corpus for English. In such corpora each word is associated with its PoS.

[('The', 'DET'), ('Fulton', 'NOUN'), ('County', 'NOUN'), ('Grand', 'ADJ'), ('Jury', 'NOUN'), ('said', 'VERB'), ('Friday', 'NOUN'), ('an', 'DET'), ('investigation', 'NOUN'), ('place', 'NOUN'), ('.', '.')]

- It learns the most frequent tag for each word in the training data.
- The result of the training is a table of two columns, the first column is a word and the second the most-frequent PoS of this word

Word	Most Frequent Tag
control	noun
run	verb
love	verb
red	adjective

Unigram Tagger

Tagging:

- During tagging, for a given word in a sentence, the tagger looks up the most likely tag associated with that word from the training data.
- If the word was seen during training, it assigns the most common tag associated with that word.
- If the word wasn't seen during training, it assigns a default tag (usually NN).

Cons:

- It doesn’t consider the context or POS tags of previous words. Consequently, a word is always tagged with the same POS, independent of its context.

Implementation steps:

- Import necessary modules.
- Load the brown corpus and divide the data into training and testing data
- Train the UnigramTagger on training data
- Tag a sentence using the tag() method of UnigramTagger.

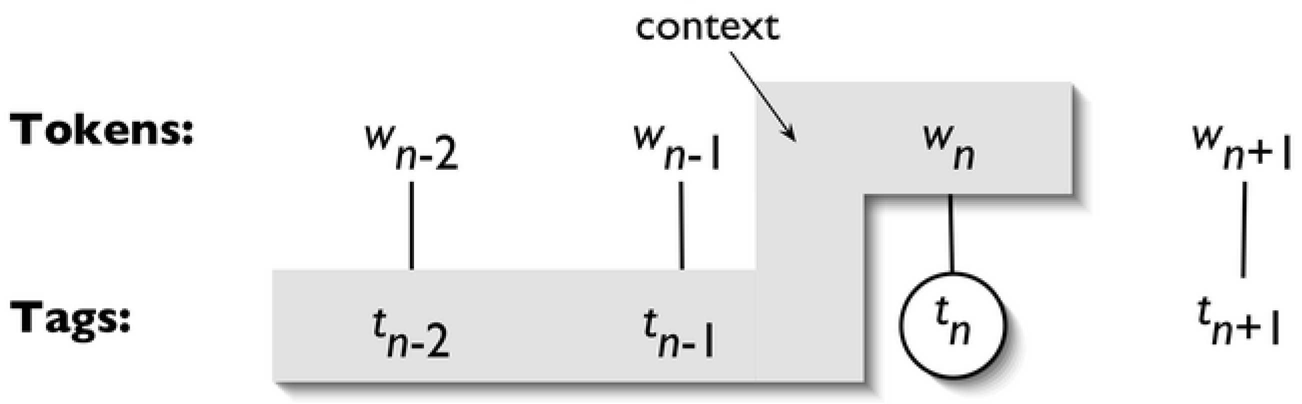
Unigram Tagger

```python
import nltk
from nltk.corpus import brown
from nltk.tag import UnigramTagger

# Download the required NLTK resources
nltk.download('brown')
nltk.download('punkt')

# Load the brown dataset for training
train_data = brown.tagged_sents()[:3000]   # Use first 3000 sentences for training
test_data = treebank.tagged_sents()[3000:]   # Use remaining sentences for testing

# Create a UnigramTagger (1-gram model)
unigram_tagger = UnigramTagger(train_data)

# Tag a sentence using the trained UnigramTagger
sentence = "The dog sat on the mat".split()
tagged_sentence = unigram_tagger.tag(sentence)

print(tagged_sentence)

print(unigram_tagger.evaluate(test_data))
```

```
[('The', 'AT'), ('dog', 'NN'), ('sat',
'VBD'), ('on', 'IN'), ('the', 'AT'),
('mat', 'NN')]

0.7641347348147013
```

N-gram tagger

- **N-gram tagger** in NLP uses the POS tags of *N-1* previous words to predict the tag for a current word.
- **N-gram-Tagger** assigns the PoS-tag of the current word by taking into account the current word itself and the PoS-tag of the N-1 preceding words.



N-gram Models:

•**Unigram Tagger (1-gram)**: It predicts the tag for the current word based on the word alone.

•**Bigram Tagger (2-gram)**: It predicts the tag for the current word based on the previous word and its tag.

•**Trigram Tagger (3-gram)**: It predicts the tag for the current word based on the previous two words and their tags.

N-gram(bigram) tagger training

The goal during training is to compute the probabilities needed for tagging. This involves:

a. Building the bigram model

- A **bigram** refers to a pair of consecutive tags, (t_{i-1}, t_i), where:
  - t_{i-1} is the tag of the previous word.
  - t_i is the tag of the current word.
- Training data consists of sentences where words are already tagged (e.g., `The/DT cat/NN sleeps/VBZ`).

From the training data:

1. **Tag Transition Probabilities**: Compute the probability of one tag following another:

$$P(t_i|t_{i-1}) = \frac{\text{Count}(t_{i-1}, t_i)}{\text{Count}(t_{i-1})}$$

   This models how likely a tag t_i is given the previous tag t_{i-1}.

2. **Emission Probabilities**: Compute the probability of a word w_i being associated with a tag t_i:

$$P(w_i|t_i) = \frac{\text{Count}(t_i, w_i)}{\text{Count}(t_i)}$$

N-gram tagger Tagging

- The tagger calculates the most probable tag sequence using the learned probabilities.
- For example:
    - In a **bigram model**, the tag t_i for a word w_i is chosen based on:

$$\text{argmax}_{t_i} P(t_i|t_{i-1}) \cdot P(w_i|t_i)$$

    - In a **trigram model**, it's based on:

$$\text{argmax}_{t_i} P(t_i|t_{i-1}, t_{i-2}) \cdot P(w_i|t_i)$$

Tag Assignment:

- The tagger assigns the tag with the highest probability to each word in the sentence.

bi-gram tagger Tagging Example

Training Data:

"The/DT cat/NN sat/VBD on/IN the/DT mat/NN

1. **Compute Emission Probabilities:** $P(\text{word}|\text{tag})$ Count occurrences of each word given a tag.

   Example:

   - $P(\text{cat}|\text{NN}) = \frac{\text{Count}(\text{cat, NN})}{\text{Count}(\text{NN})} = \frac{1}{2}$
   - $P(\text{sat}|\text{VBD}) = \frac{1}{1} = 1.0$

2. **Compute Transition Probabilities:** $P(\text{current tag}|\text{previous tag})$ Count transitions between tags.

   Example:

   - $P(\text{NN}|\text{DT}) = \frac{\text{Count}(\text{DT} \rightarrow \text{NN})}{\text{Count}(\text{DT})} = \frac{2}{2} = 1.0$
   - $P(\text{VBD}|\text{NN}) = \frac{\text{Count}(\text{NN} \rightarrow \text{VBD})}{\text{Count}(\text{NN})} = \frac{1}{2} = 0.5$

Tagging Phase:

$$P(\text{tag}_t|\text{word}_t, \text{tag}_{t-1}) = P(\text{word}_t|\text{tag}_t) \times P(\text{tag}_t|\text{tag}_{t-1})$$

Ex: the cat is running

For word cat:

P(cat|NN)×P(NN|DT)=0.5x 1.0
=0.5

P(cat|VBD)xP(VBD|DT)= 0x0
=0

NN tag for cat having highest probabilities so assign tag NN to cat

```python
import nltk
from nltk.corpus import brown
from nltk.tag import UnigramTagger, BigramTagger, TrigramTagger

nltk.download('brown')
nltk.download('punkt')

train_data = brown.tagged_sents()[:3000]  # First 3000 sentences for training
test_data = brown.tagged_sents()[3000:]   # Remaining sentences for testing

# Create and train a Unigram Tagger
unigram_tagger = UnigramTagger(train_data)

# Create and train a Bigram Tagger
bigram_tagger = BigramTagger(train_data, backoff=unigram_tagger)

# Create and train a Trigram Tagger
trigram_tagger = TrigramTagger(train_data, backoff=bigram_tagger)

# Test the trigram tagger on a new sentence
sentence = "The dog sat on the mat".split()
tagged_sentence = trigram_tagger.tag(sentence)

print(tagged_sentence)

# Print Performnces
print(unigram_tagger.evaluate(test_data))
print(bigram_tagger.evaluate(test_data))
print(trigram_tagger.evaluate(test_data))
```

```
[('The', 'AT'), ('dog', 'NN'), ('sat', 'VBD'),
('on', 'IN'), ('the', 'AT'), ('mat', 'NN')]
```

```
0.7641347348147013
0.7734657846307614
0.772435283624883
```

Examples

<S> I like to play with it </S>
<S> You like to play </S>

Total count of unigrams (without <S> and </S> tags) :10

Total count of bigrams (including <S> and </S> tags): 12

`<S> I`, `I like`, `like to`, `to play`, `play with`, `with it`, `it </S>`

`<S> You`, `You like`, `like to`, `to play`, `play </S>`

Unigram probabilities for: P(like) =

$$P(\text{word}) = \frac{\text{Count(word)}}{\text{Total count of unigrams}}$$

P(like):

- Count of "like" = 2
- Total unigrams = 10

$$P(\text{like}) = \frac{2}{10} = 0.2$$

Examples

<S> I like to play with it </S>
<S> You like to play </S>

Conditional probabilities: P(like | it)

Conditional probability:

$$P(w_2|w_1) = \frac{\text{Count}(w_1 \to w_2)}{\text{Count}(w_1)}$$

- $P(\text{like}|\text{it}) = \frac{\text{Count}(\text{it}\to\text{like})}{\text{Count}(\text{it})} = \frac{0}{1} = 0.$

Bigram probability for: P(you like)

$$P(w_2|w_1) = \frac{\text{Count}(w_1 \to w_2)}{\text{Count}(w_1)}$$

- $P(\text{you like}) = \frac{\text{Count}(\text{you}\to\text{like})}{\text{Count}(\text{you})} = \frac{1}{1} = 1.0.$

Trigram probability for: P(you like it)

Trigram probability:

$$P(w_3|w_1, w_2) = \frac{\text{Count}(w_1 \to w_2 \to w_3)}{\text{Count}(w_1 \to w_2)}$$

P(you like it):

- Count of ("You like it") = 0 (No trigram "You like it")
- Count of ("You like") = 1

$$P(\text{you like it}) = \frac{0}{1} = 0$$

N-gram

Pros of N-gram Tagger

- Captures Local Context
- Improves Predictions
- Easy to Implement
- Flexibility in N-gram Size

Cons of N-gram Tagger:

Requires Large Amount of Training Data:

•N-gram models, especially for higher values of N, require large amounts of labeled data to accurately estimate the probabilities of sequences.

Memory Intensive:

•Higher-order N-grams (trigrams, 4-grams, etc.) can become memory-intensive because they require storing a large number of tag combinations

Lack of Flexibility:

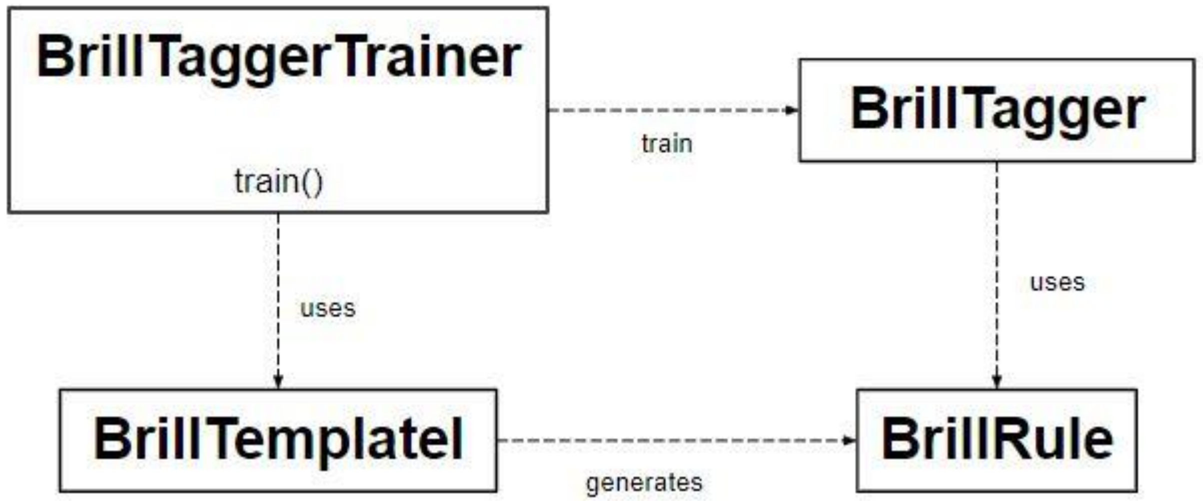
•N-gram taggers are not able to capture global syntactic dependencies or hierarchical structures in sentences.

Brill tagger

- The **Brill Tagger** is a **transformation-based tagger** introduced by Eric Brill.
- It uses an initial tagger (unigram tagger) and then applies transformation rules iteratively to correct errors made by the initial tagger.
- Moreover, it uses a series of rules to correct the results of an initial tagger.

•These rules it follows are scored based.

This score is equal to the no. of errors they corrected minus the no. of new errors they produced.



Training the Brill Tagger

a. Initial Tagging

•Apply an initial tagger, which can be:

- A simple default tagger or statistical tagger (e.g., unigram or bigram tagger) to assign tags initially.

b. Learning Transformation Rules

1.Define a Template:

1. Specify templates for rules, such as:
    1. Change the tag of a word if the previous word is a specific tag.
    2. Change the tag of a word if the next word is a specific tag.

    Ex: Change NN to VB if the previous word is a noun

2.Error Analysis:

- Compare the initial tagging results with the ground truth from the training data to identify tagging errors.

3.Rule Generation, Evaluation & Selection

- Generate potential rules from the template to correct the errors.
- Score each rule by no. of errors they corrected minus the no. of new errors they produced.
- Select the best rule based on its score and add it to the list of transformation rules.

4.Iterative Process:

1. Apply the selected rule to the training data and repeat the process until:
    1. A stopping condition is met (e.g., no significant improvement, maximum number of rules reached).

Testing the Brill Tagger

a. Apply Initial Tagging

- Use the same initial tagger as in the training phase to assign initial tags to the testing set.

b. Apply Learned Rules

- Apply the learned transformation rules, in the same order as learned, to refine the initial tags.

c. Evaluate Performance

- Compare the resulting tags with the ground truth annotations in the test set.

```python
import nltk
from nltk.tag import brill, brill_trainer, UnigramTagger
from nltk.corpus import brown

nltk.download('brown')

train_data = brown.tagged_sents()[:3000]
test_data = brown.tagged_sents()[3000:]

# Step 2: Define the Initial Tagger
initial_tagger = UnigramTagger(train_data)

# Step 3: Define Brill Tagger Templates
templates = brill.fntbl37()

# Step 4: Train the Brill Tagger
trainer = brill_trainer.BrillTaggerTrainer(initial_tagger, templates)
brill_tagger = trainer.train(train_data)

sentence = "The dog sat on the mat."
tokenized_sentence = nltk.word_tokenize(sentence)
tagged_sentence = brill_tagger.tag(tokenized_sentence)

print(tagged_sentence)

# Evaluate Brill Tagger
accuracy = brill_tagger.evaluate(test_data)
print(f"Accuracy of the Brill Tagger: {accuracy:.2f}")
```

```
[('The','AT'),('dog','NN'),('sat', 'VBD'),
('on', 'IN'), ('the', 'AT'), ('mat',
'NN'), ('.', '.')]
```

```
0.79
```

Machine learning taggers

- HMM
- MEC
- CRF

Hidden Markov Model (HMM)

- The Hidden Markov Model (HMM) is a probabilistic sequence model used in POS tagging.
- It assigns the most likely sequence of POS tags to a sentence based on observed words.
- The components of HMMs are

States

•Represents the possible **POS tags** (e.g., noun, verb, adjective, etc.).

Observations

•Represents the **words in the sentence**

Transition Probability

- Represents the probability of one POS tag following another in a sentence

Emission Probability

- Represents the probability of a word being associated with a particular POS tag

Hidden Markov Model (HMM)

Transition Probability

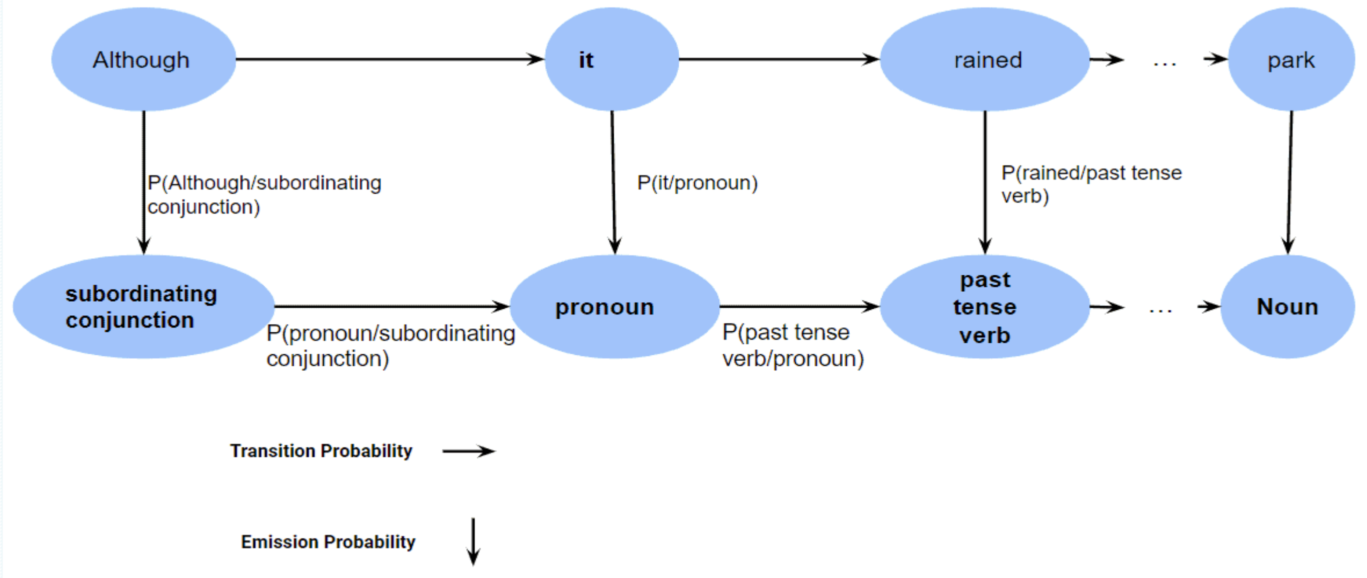
- Represents the probability of one POS tag following another in a sentence

$$P(t_i|t_{i-1}) = \frac{C(t_{i-1}, t_i)}{C(t_{i-1})}$$

Emission Probability

- Represents the probability of a word being associated with a particular POS tag

$$P(w_i|t_i) = \frac{C(t_i, w_i)}{C(t_i)}$$


Although
it
rained
...
park
P(Although/subordinating conjunction)
P(it/pronoun)
P(rained/past tense verb)
subordinating conjunction
pronoun
past tense verb
...
Noun
P(pronoun/subordinating conjunction)
P(past tense verb/pronoun)
Transition Probability
Emission Probability

Hidden Markov Model (HMM)

Q: Set of possible Tags (hidden states)

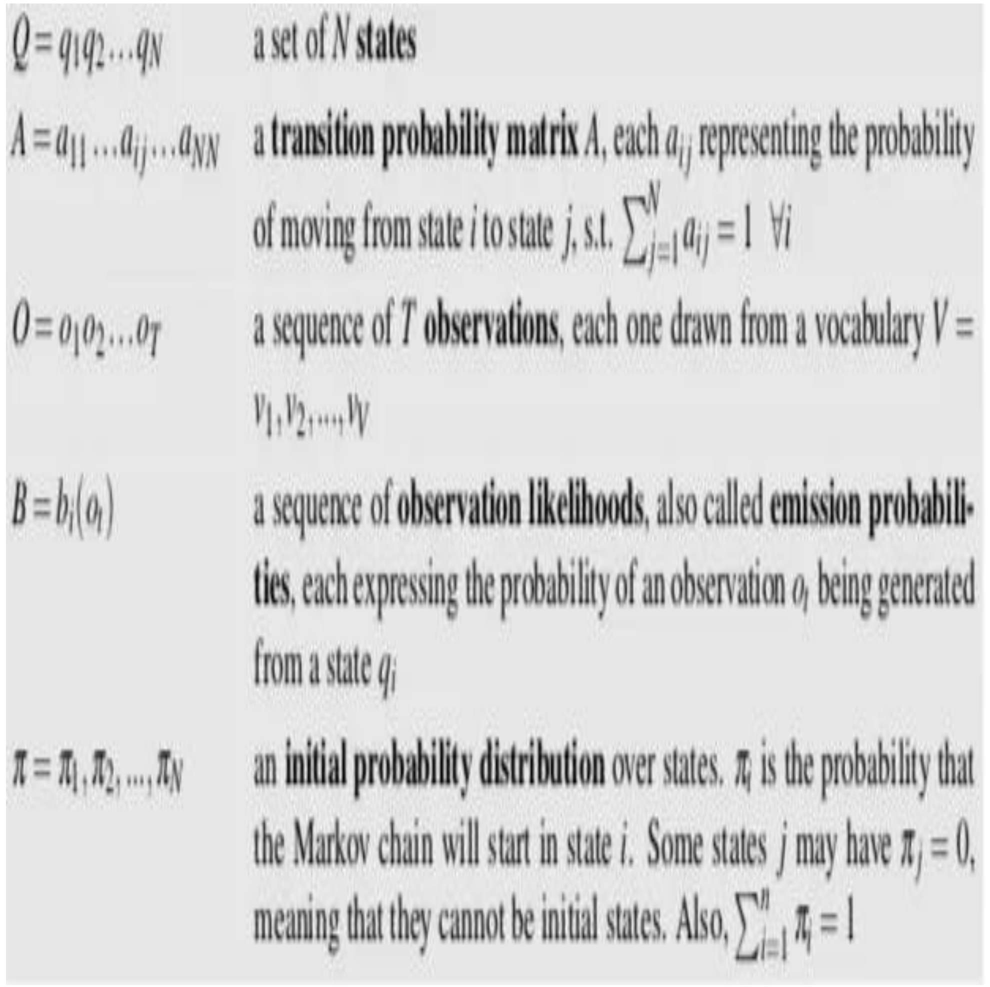
A: The A matrix contains the tag transition probabilities P(ti|ti−1) which represent the probability of a tag occurring given the previous tag.
Example: Calculating A[Verb][Noun]:

O: Sequence of observation (words in the sentence)

B: The B emission probabilities, P(wi|ti), represent the probability, given a tag (say Verb), that it will be associated with a given word (say Playing). The emission probability B[Verb][Playing] is calculated using:

P(Playing | Verb): Count (Playing & Verb)/ Count (Verb)

It must be noted that we get all these Count() from the corpus itself used for training.



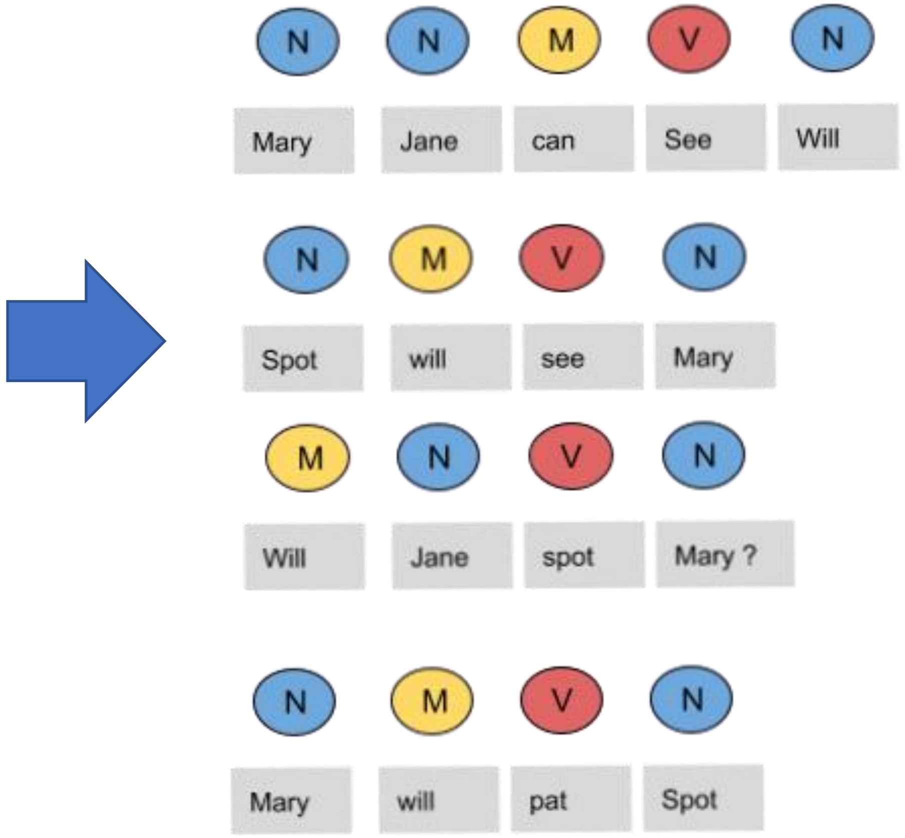
$Q = q_1 q_2 \dots q_N$	a set of N **states**
$A = a_{11} \dots a_{ij} \dots a_{NN}$	a **transition probability matrix** A, each a_{ij} representing the probability of moving from state i to state j, s.t. $\sum_{j=1}^{N} a_{ij} = 1 \ \forall i$
$O = o_1 o_2 \dots o_T$	a sequence of T **observations**, each one drawn from a vocabulary $V = v_1, v_2, \dots, v_V$
$B = b_i(o_t)$	a sequence of **observation likelihoods**, also called **emission probabilities**, each expressing the probability of an observation o_t being generated from a state q_i
$\pi = \pi_1, \pi_2, \dots, \pi_N$	an **initial probability distribution** over states. π_i is the probability that the Markov chain will start in state i. Some states j may have $\pi_j = 0$, meaning that they cannot be initial states. Also, $\sum_{i=1}^{n} \pi_i = 1$

Example

Let us calculate the above two probabilities for the set of sentences below

Mary Jane can see Will
Spot will see Mary
Will Jane spot Mary?
Mary will pat Spot

Note that Mary Jane, Spot, and Will are all names.



calculate the emission probabilities

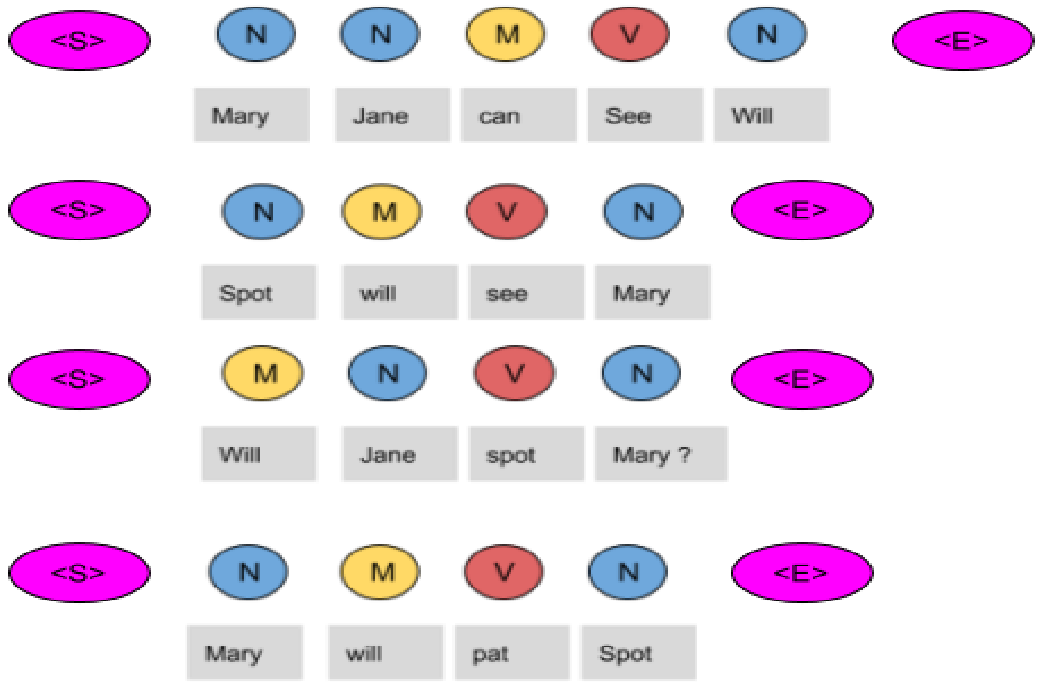
counting table

Words	Noun	Model	Verb
Mary	4	0	0
Jane	2	0	0
Will	1	3	0
Spot	2	0	1
Can	0	1	0
See	0	0	2
pat	0	0	1

emission probabilities

Words	Noun	Model	Verb
Mary	4/9	0	0
Jane	2/9	0	0
Will	1/9	3/4	0
Spot	2/9	0	1/4
Can	0	1/4	0
See	0	0	2/4
pat	0	0	1

Transition Probabilities



counting table

	N	M	V	<E>
<S>	3	1	0	0
N	1	3	1	4
M	1	0	3	0
V	4	0	0	0

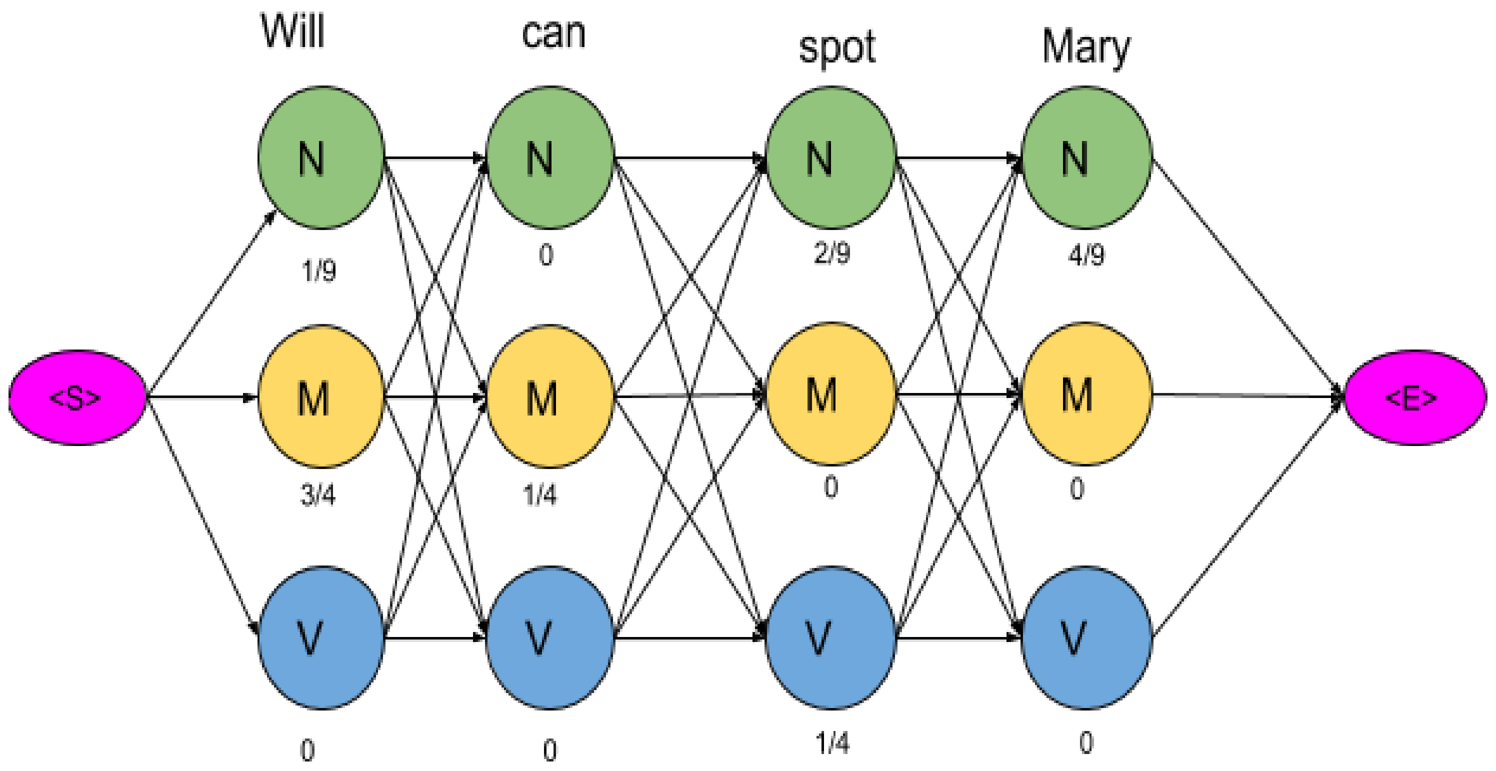
transition probabilities

	N	M	V	<E>
<S>	3/4	1/4	0	0
N	1/9	3/9	1/9	4/9
M	1/4	0	3/4	0
V	4/4	0	0	0

POS tagging for new sentence

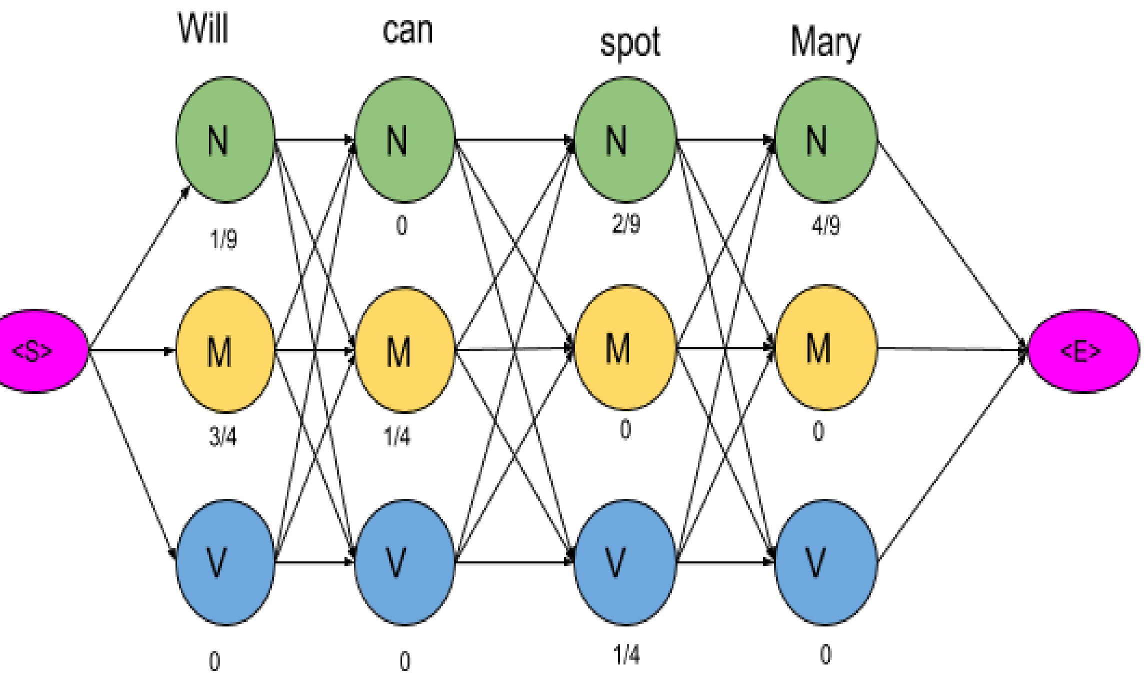
Sentence: Will can spot Mary

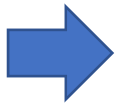
All possible combinations with three tags(N, M and V)

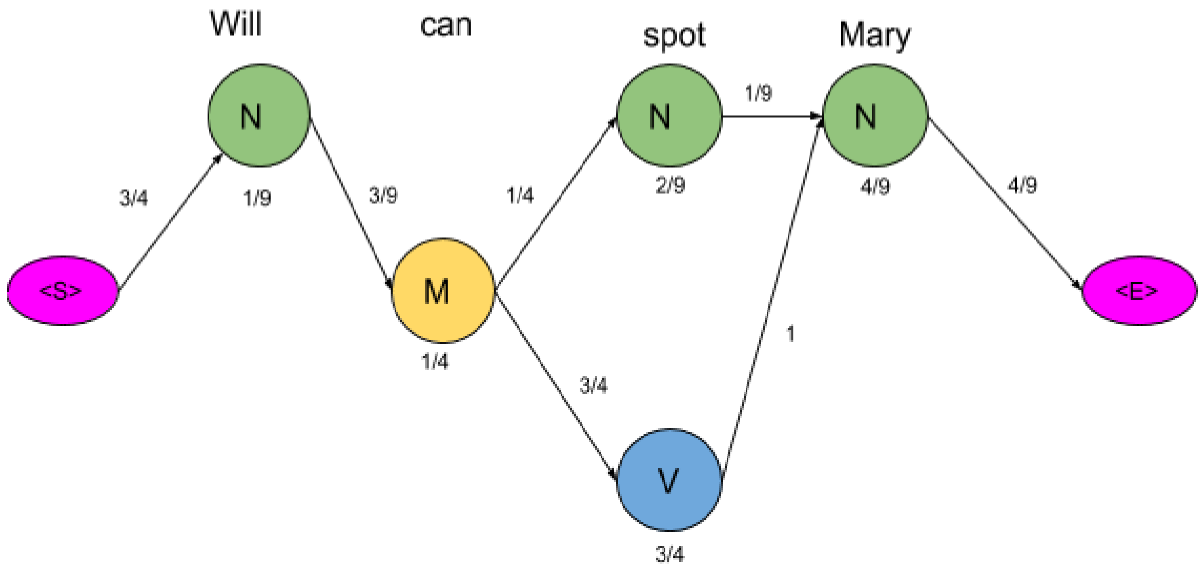


POS tagging for new sentence

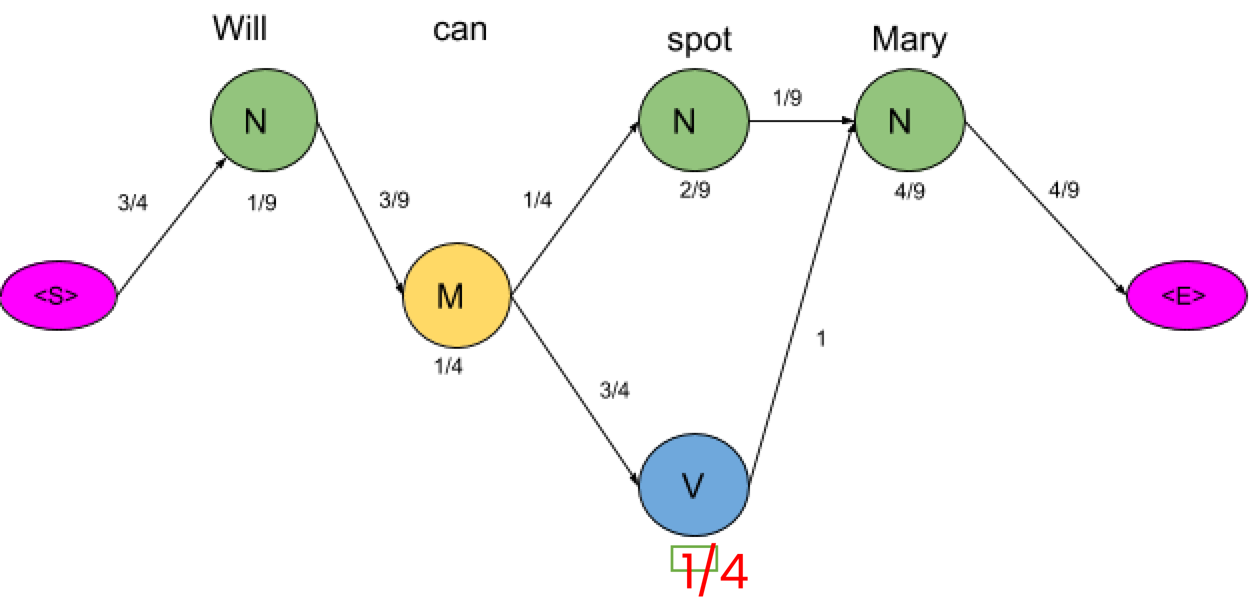
The next step is to delete all the vertices and edges with probability zero, also the vertices which do not lead to the endpoint are removed.







POS tagging for new sentence



<S>→N→M→N→N→<E> =**3/4*1/9*3/9*1/4*1/4*2/9*1/9*4/9*4/9=0.00000846754**
<S>→N→M→V→N→<E>=**3/4*1/9*3/9*1/4*3/4*1/4*1*4/9*4/9=0.00025720164**

Clearly, the probability of the second sequence is much higher and hence the HMM is going to tag each word in the sentence according to this sequence.

Will as a Noun
Can as a Model
Spot as a Verb
Mary as a noun

Working of HMM tagger

1. Training Phase

In this phase, we estimate the parameters of the HMM model using a labeled corpus (e.g., Penn Treebank). The parameters include **transition probabilities** and **emission probabilities**.

Step 1: Preprocess the Training Data

- Tokenize the text into sentences and words.
- Each token has a corresponding POS tag.

Step 2: Estimate Transition Probabilities (A)

Transition probabilities represent the likelihood of a POS tag given the previous tag:

$$P(t_i|t_{i-1}) = \frac{C(t_{i-1}, t_i)}{C(t_{i-1})}$$

where:

- $C(t_{i-1}, t_i)$ is the count of the bigram (t_{i-1}, t_i) in the training data.
- $C(t_{i-1})$ is the count of tag t_{i-1}.

Step 3: Estimate Emission Probabilities (B)

Emission probabilities represent the likelihood of a word given a tag:

$$P(w_i|t_i) = \frac{C(t_i, w_i)}{C(t_i)}$$

where:

- $C(t_i, w_i)$ is the count of word w_i occurring with tag t_i.
- $C(t_i)$ is the count of tag t_i.

Working of HMM tagger

2. Testing (POS Tagging Phase)

- Given a new sentence, the trained HMM is used to predict the sequence of POS tags using the **Viterbi algorithm**.

$$v_t(j) = \max_{i=1}^{n} v_{t-1}(i) a_{ij} b_j(o_t)$$

$v_{t-1}(i)$:	Viterbi path probability from the previous time
a:	(Transition matrix)
b:	(Emission matrix)

Code of HMM tagger

```python
import nltk
from nltk.corpus import treebank
from nltk.tag import hmm

nltk.download('treebank')
nltk.download('universal_tagset')

# Prepare the data
train_sents = treebank.tagged_sents(tagset='universal')
test_sents = treebank.tagged_sents(tagset='universal')[3000:]

# Train the HMM tagger
trainer = hmm.HiddenMarkovModelTrainer()
hmm_tagger = trainer.train(train_sents)

sentence = "This is a test sentence".split()

# Tag the sentence
tagged_sentence = hmm_tagger.tag(sentence)
print(tagged_sentence)
```

Disadvantages of HMM tagger

Inability to Handle Unknown Words (OOV - Out of Vocabulary Words)

- If a word is not present in the training corpus, the model struggles to assign a correct POS tag.

Limited Context Awareness

- HMM can only use **previous** tags for prediction.
- Cannot consider future words while tagging the current word

Data Sparsity Problem

- If a word-tag combination is rare or absent, the model performs poorly.

Computational Inefficiency for Large Tagsets

Conditional Random Fields (CRF) POS tagging

- Conditional Random Fields (CRF) for Part-of-Speech (POS) tagging is a sequence labeling technique that assigns a POS tag to each word in a sentence while considering contextual dependencies.
- It is particularly useful because it captures contextual dependencies between words, unlike traditional classifiers that treat each word independently.

Why Use CRF for POS Tagging?

•**Sequence Dependencies**: Unlike Hidden Markov Models (HMMs), CRF does not assume independence between features.

•**Feature Flexibility**: Can incorporate various linguistic features such as word suffixes, capitalization, and surrounding words.

•**Handles Ambiguity Well**: Unlike simple classifiers (e.g., SVM, Decision Trees), CRF considers the entire sentence while assigning POS tags.

Conditional Random Fields (CRF) POS tagging

1. Training Phase

- The training phase involves learning the parameters (weights) of the CRF model from labeled data.

Step 1: Preparing the Training Data

•The dataset consists of sentences where each word is labeled with its corresponding POS tag.

Example training sentence:

Sentence: ["The", "cat", "sits"]

POS Tags: ["DT", "NN", "VBZ"]

Step 2: Feature Extraction

- For each word in a sentence, extract hand-crafted features such as:

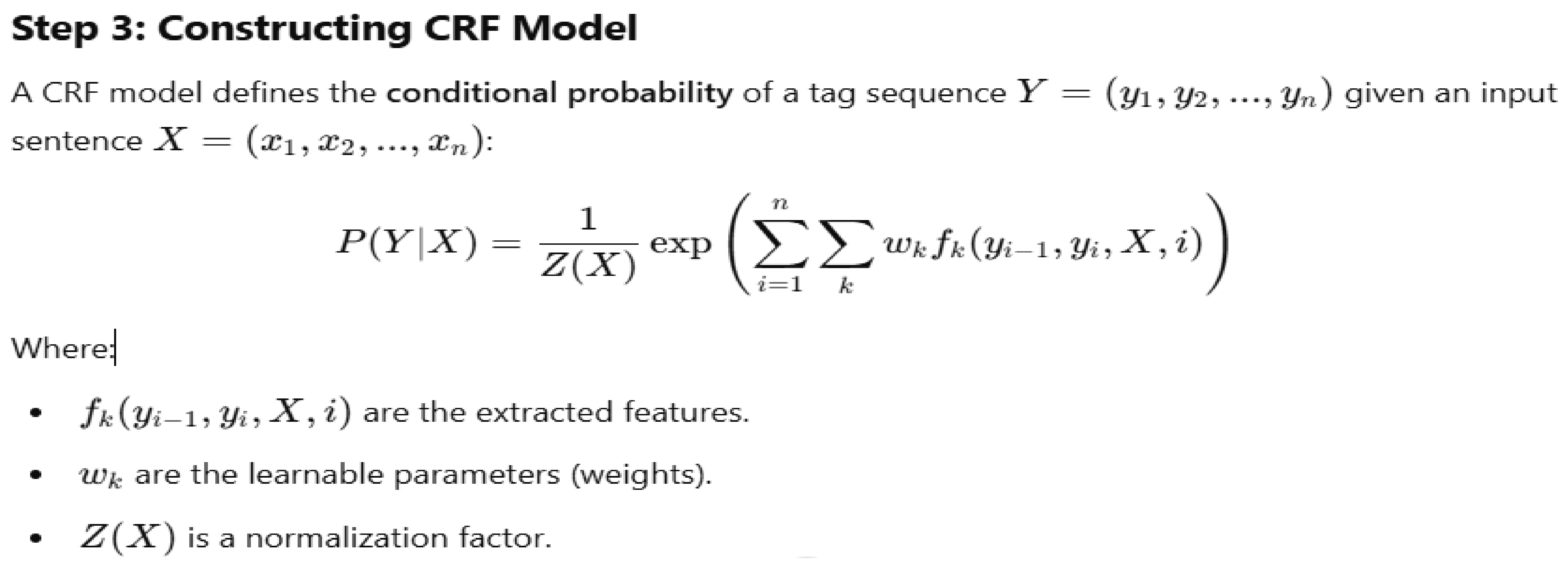
Word identity: w_i = "cat“

Previous word: w_{i-1} = "The“

Next word: w_{i+1} = "sits“

Word shape: "Xx" (for capitalized words)

Prefixes/Suffixes: ca-, -at



Step 3: Constructing CRF Model

A CRF model defines the **conditional probability** of a tag sequence $Y = (y_1, y_2, \ldots, y_n)$ given an input sentence $X = (x_1, x_2, \ldots, x_n)$:

$$P(Y|X) = \frac{1}{Z(X)} \exp\left(\sum_{i=1}^{n} \sum_{k} w_k f_k(y_{i-1}, y_i, X, i)\right)$$

Where:

- $f_k(y_{i-1}, y_i, X, i)$ are the extracted features.
- w_k are the learnable parameters (weights).
- $Z(X)$ is a normalization factor.

Step 4: Learning Parameters

•The goal is to find the best weights w_k that maximize the likelihood of the correct POS sequence.

•We use an optimization algorithm like **Gradient Descent** or **L-BFGS** to adjust the weights based on the training data.

Conditional Random Fields (CRF) POS tagging

2. Testing (Inference) Phase

During testing, we predict POS tags for new, unseen sentences.

Step 1: Feature Extraction

- Extract features from the input sentence just like in training.

Step 2: Predicting the Best Tag Sequence

- Since CRF considers the entire sentence instead of making independent predictions per word, **Viterbi Algorithm** (dynamic programming) is used to find the most probable sequence of tags Y^*.

$$Y^* = \arg\max_{Y} P(Y|X)$$

```
import sklearn_crfsuite
from sklearn_crfsuite import metrics

# Extract features for each word
def word_features(sentence, index):
    word = sentence[index][0]
    features = {
        'word': word,
        'is_first': index == 0,
        'is_last': index == len(sentence) - 1,
        'is_title': word.istitle(),
        'is_upper': word.isupper(),
        'is_digit': word.isdigit(),
        'prev_word': '' if index == 0 else
                        sentence[index - 1][0],
        'next_word': '' if index == len(sentence) - 1
                        else sentence[index + 1][0],
        'prefix-1': word[:1],
        'prefix-2': word[:2],
        'suffix-1': word[-1:],
        'suffix-2': word[-2:],
    }
    return features

# Convert dataset to feature format
def sentence_features(sentence):
    return [word_features(sentence, i) for i in
range(len(sentence))]

def sentence_labels(sentence):
```

```
# Sample training dataset
train_data = [
    [('The', 'DET'), ('cat', 'NOUN'), ('sat',
'VERB'), ('on', 'PREP'), ('the', 'DET'), ('mat',
'NOUN')],
    [('A', 'DET'), ('dog', 'NOUN'), ('barked',
'VERB')],
]

X_train = [sentence_features(sentence) for sentence
in train_data]
y_train = [sentence_labels(sentence) for sentence in
train_data]

# Train CRF model
crf = sklearn_crfsuite.CRF(algorithm='lbfgs')
crf.fit(X_train, y_train)


test_sentence = [('The'), ('man'),('running')]
X_test = [sentence_features(test_sentence)]
y_pred = crf.predict(X_test)
print(y_pred[0])  # Predicted POS tags
```

```
['DET' 'NOUN' 'VERB']
```

NER Tagger

- Named Entity Recognition (NER) is a NLP technique to find and classify entities from textual data into predefined categories called named entities.

Types of Named Entities:

Person: Names of individuals (e.g., *Elon Musk*, *Marie Curie*).
Organization: Names of organizations (e.g., *Google*, *United Nations*).
Location: Geographical entities (e.g., *Paris*, *Mount Everest*).
Date/Time: Temporal expressions (e.g., *January 1, 2025*, *10:30 AM*).
Monetary values: Financial amounts (e.g., *$10,000*).
Percentages: Percentage figures (e.g., *50%*).
Miscellaneous: Other domain-specific categories (e.g., product names, scientific terms).

- A Named Entity Recognition (NER) tagger is a tool used in NLP to identify and classify entities within a text.
- Named entity recognition is important because it enables organizations to extract valuable information from unstructured text data.
- For example, an NER system could be used to extract the names of all the companies mentioned in a set of news articles, along with their stock prices, market capitalization, and other relevant attributes.

NER using Spacy library

```python
import spacy

# Load the pre-trained SpaCy model
nlp = spacy.load("en_core_web_sm")

# Define the text
text = """
Elon Musk, the CEO of SpaceX, announced that the company will launch a new mission to Mars
in 2025. The announcement was made in Paris on January 20, 2025, and the mission will cost
around $10,000,000. The United Nations has also shown interest in the project, especially
due to its potential to address climate change. Approximately 50% of the project's funding
will come from private investors.
"""

# Process the text
doc = nlp(text)

#print entities
print(doc.ents)

# Iterate over the entities detected in the text
for ent in doc.ents:
    print(f"Text: {ent.text}, Label: {ent.label_}")
```

```
(Elon Musk, SpaceX, Mars, 2025, Paris, January
20, 2025, around $10,000,000, The United Nation
Approximately 50%)

Elon Musk: PERSON
SpaceX: GPE
Mars: LOC
2025: DATE
Paris: GPE
January 20, 2025: DATE
around $10,000,000: MONEY
The United Nations: ORG
Approximately 50%: PERCENT
```

Applications of NER

Information Extraction:

- Extracting structured information from unstructured text (e.g., extracting dates, names, and locations from news articles).

Search Engine Optimization:

- Enhancing search relevance by identifying key entities in user queries.

Document Categorization:

- Automatically categorizing documents based on detected entities.

Question Answering Systems:

- Understanding user queries and identifying key entities.

Customer Support:

- Recognizing entities like product names or customer details in support tickets.