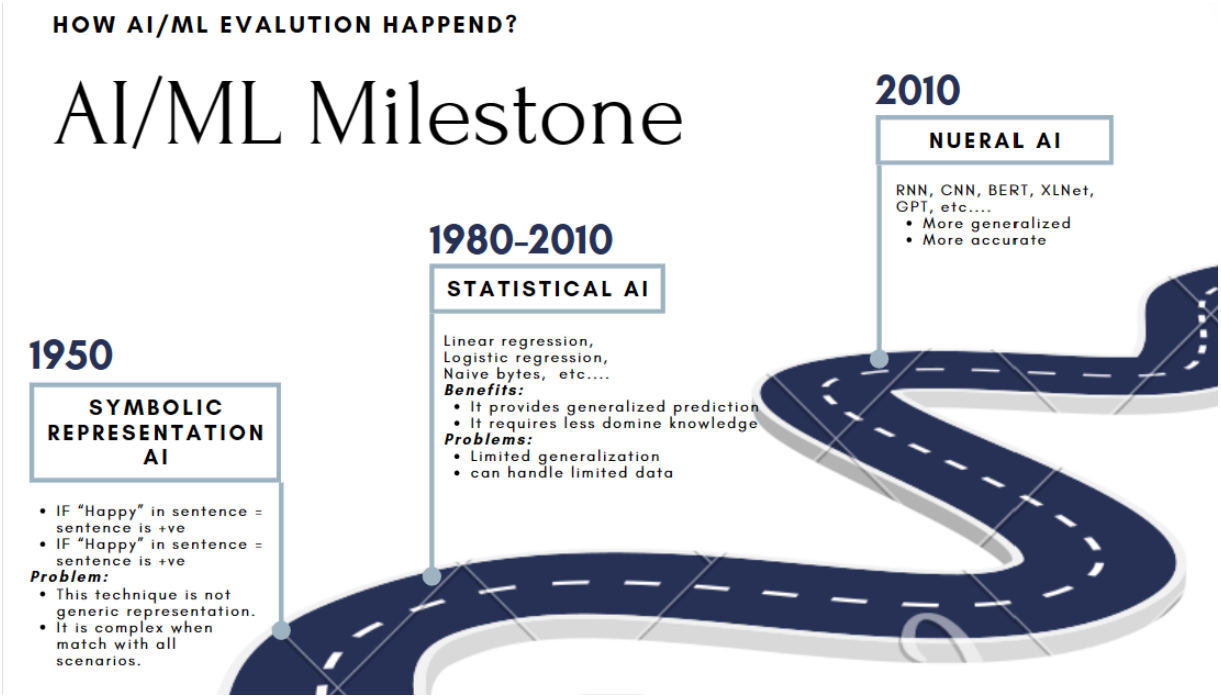
Introduction to NLP

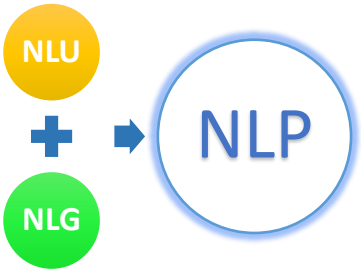
The NLP stands for ***Natural Language Processing***. it is one of the branches of artificial intelligence, that can understand or speak or write the human language.

Mainly two types of python libraries used for NLP, which are ***NLTK*** and ***Spacy.***



1. NLP Branches:

The NLP has two branches are NLU and NLG:



1.1. NLU – (Natural Language Understanding)

- NLU is a *computer program / NLP algorithm* that can understand human language in the form of voice records or text.
- Here the natural language providing as input to NLU and it will understand the input and will give a response in natural language according to our input.

1.2. NLG – (Natural Language Generation)

- NLG is a computer program that can generate human language in the form of speech or text based on our request.
- NLG takes our requests as input and generates the natural language that can understandable by human.

2. NLP Stages:

NLP divided into four stages. These stages are very important to create accurate model. Let's learn one by one below.

1. Tokenization
2. Data cleaning
3. Vectorization / Word embedding
4. Model Development

3. Tokenization

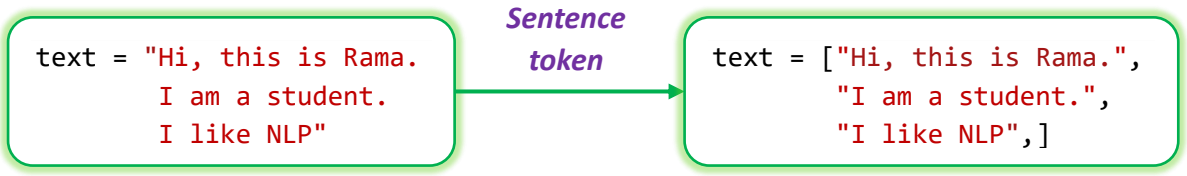
- In NLP Tokenization is the first step. In this stage the whole text will divided into meaningful units are called the tokens.
- The NLP or ML algorithms can understand the text in the form vector only. So, to convert a text into vectors, first we should generate the tokens of that text and will convert those tokens into vectors. This is the main use of tokenization.

There are three types of tokenization techniques are using based on the requirement:

1. Sentence token
2. Tokens of sentences
3. Tokens of whole text

3.1. Sentence token

- The *sentence token* is a technique that converts each sentence as a token from whole text.
- In this stage we can remove all non-required special characters from each sentence.



By Using NLTK:

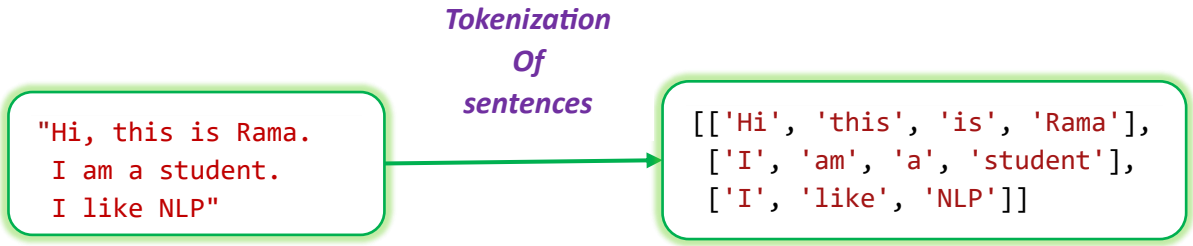
```
1. import nltk

2. text = "Hi, this is Rama. I am a student. I like NLP" # Input text

3. # Extract sentences from the processed document
4. sent_tokens = nltk.sent_tokenize(text)
5. print(sent_tokens)

# OUTPUT: ['Hi, this is Rama.', 'I am a student.', 'I like NLP']
```

By Using SpaCy:

```
1. import spacy

2. nlp = spacy.load('en_core_web_sm') # Load the English language model
3. doc = nlp(text) # Process the text with spaCy

4. # Extract sentences from the processed document
5. sentences = [sent.text for sent in doc.sents]
6. print(sentences)

# OUTPUT: ['Hi, this is Rama.', 'I am a student.', 'I like NLP']
```

3.2. Tokens of Sentences

- The *tokens of sentences* is nothing but process converting each sentence into tokens and each sentence tokens will be a list.



By Using NLTK:

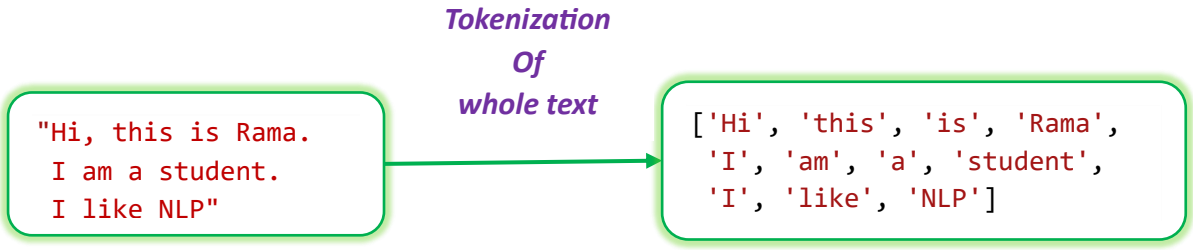
```
1. tokens_of_sents = [list(nltk.word_tokenize(sent)) for sent in
   sent_tokens]
2. print(tokens_of_sents)

# OUTPUT: [['Hi', ',', 'this', 'is', 'Rama', '.'], ['I', 'am', 'a', 'student', '.'], ['I', 'like', 'NLP']]
```

By Using SpaCy:

```
1. import spacy

2. nlp = spacy.load('en_core_web_sm') # Load the English language model
3. doc = nlp(text) # Process the text with spaCy

4. # Extract tokens of sentences from the processed document
5. tokens_of_sents = [[token for token in sent] for sent in doc.sents]
6. print(tokens_of_sents)

# OUTPUT: [[Hi, ,, this, is, Rama, .], [I, am, a, student, .], [I, like, NLP]]
```

3.3. Tokens of Whole Text

- The *tokens of whole text* is nothing but process converting whole text into a list of tokens.



By Using NLTK:

```
1. tokens_of_text = nltk.word_tokenize(text)
2. print(tokens_of_text)

# OUTPUT:  [Hi, ,, this, is, Rama, ., I, am, a, student, ., I, like, NLP]
```

By Using SpaCy:

```
1. import spacy

2. nlp = spacy.load('en_core_web_sm') # Load the English language model
3. doc = nlp(text) # Process the text with spaCy

4. # Extract tokens of text from the processed document
5. tokens_of_text = [token for token in doc]
6. print(tokens_of_text)

# OUTPUT: [Hi, ,, this, is, Rama, ., I, am, a, student, ., I, like, NLP]
```

4. Data Cleaning

Data cleaning is a very important step to generate affordable text to create vectorization format. Let's start learn all data cleaning topics here.

Steps involved in data cleaning:

1. Remove stop-words
2. Convert to lower case
3. Stemming
4. Lemmatization
5. Removing special and hidden characters.

Consider the below text as example input natural language sentence to perform the data cleaning:

```
text = 'I am in America, Working as a DATA SCIENTIST and reporting @ 10:00 am every day…!'
```

4.1. Removing Stop-Words

- Stop-words are some simple and commonly used words in human languages to form a sentence. ***Ex:*** *am, in, as, a, an, and…etc.*
- These stop-words don't create any sense and don't have any extra value once the text converted into vector format.
- Sometimes there are some words not important to business case then those words are also treated as stop words.

By Using NLTK:

```
1. from nltk.corpus import stopwords
2. # Input text
3. text = 'I am in America, Working as a DATA SCIENTIST and urgent to reporting @10:00 am every day…!'
4. useless_wrds = ['urgent', 'may', 'can'] # Use-less words
5.
6. # Adding useless_wrds to stop words
7. final_stopwords = stopwords.words('english')
```

```
final_stopwords.extend(useless_wrds)

# Removing stop words
final_text = ' '.join([word for word in text.split() if word not in final_stopwords])
print(final_text)

# OUTPUT: I America, Working DATA SCIENTIST reporting @10:00 every day...!
```

By Using SpaCy:

```
import spacy

nlp = spacy.load("en_core_web_sm") # Loading the nlp model
# Creating document from nlp model
doc = nlp('I am in America, Working as a DATA SCIENTIST and urgent to reporting @10:00 am every day…!')

useless_wrds = ['urgent', 'may', 'can'] # Use-less words

# Final text after remove stop words
print(" ".join([token.text for token in doc if (not token.is_stop) and (not token.text in useless_wrds) ]))

# OUTPUT: America , Working DATA SCIENTIST reporting @10:00 day ... !
```

4.2. Convert To Lower Case

- The conversion of text to lower case is very important step to maintain the uniformity. Some types of lowercase conversion benefits mentioned in below.
- ***Normalization:*** Lowercasing the text helps normalize the input by reducing the number of unique word variations. For example, "Help," "help," and "HELP" are essentially the same word, and converting them all to lowercase ensures consistency and reduces the vocabulary size
- ***Word Matching:*** Lowercasing text allows for better word matching and comparison.
- ***Vocabulary Size:*** Lowercasing helps reduce the vocabulary size by collapsing words with different capitalization into a single entry.
- ***Generalization:*** Lowercasing text helps with generalization by treating different cases of the same word as equivalent.

```
# Input text
text = 'I am in America, Working as a DATA SCIENTIST and urgent to reporting @10:00 am every day…!'

final_text = text.lower()# Converting text into lower case.
print(final_text)
```

```
# OUTPUT: i am in america, working as a data scientist and urgent to reporting @10:00 am every day...!
```

4.3. Stemming

- Stemming is the process of cutting suffix part of the word and gives it's root word called "stem".
- The purpose of stemming is to normalize words so that different variations of the same word are treated as one, thereby it reducing the dimensionality of the vocabulary and improving the efficiency of text analysis.
  Ex: For example, consider the words "running," "runs," and "ran." These words all share the same root concept of "run." After applying stemming, they would all be reduced to their common stem, "run".

Some times stemming leads to create incorrect word by blindly remove suffixes. Stemming generates **"runn"** from "**running**" but the correct root word is **"run".** This problem can resolve buy lemmatization concept.

By Using NLTK:

```
1. from nltk.stem import PorterStemmer

2. # Input text
3. text = 'i am in america, working as a data scientist and urgent to
   reporting @10:00 am every day…!'

4. post_stemmer = PorterStemmer()# Generate the stemming.
5. final_text = ' '.join([post_stemmer.stem(word) for word in
   text.split()])

6. print(final_text)
```

```
# OUTPUT: i am in america, work as a data scientist and urgent to report @10:00 am everi day...!
```

Note: Spacy not supporting the stemming concept. We have only NLTK library to achieve this

4.4. Lemmatization

- Limitation is advanced technique of stemming; it overcomes the root word in character recognition problem.
- It generates the lemma of the word instead of blindly removing the suffix. Lemmatization is the best technique than compared to stemming.

Lemmatization doesn't work properly in spaCy after removing stop words. Lemmatization is the process of reducing words to their base or root form, and it depends on the context of the words in a sentence. Removing stop words may lead to changes in the context, and that can affect lemmatization results.

By Using NLTK:

```
1. from nltk.stem import wordnet, WordNetLemmatizer
2. from nltk.tokenize import word_tokenize

3. # Input text
4. text_data = 'i am in america, works as a data scientist and urgent to
   reporting @10:00 am every day…!'

5. lem = WordNetLemmatizer()# Assigning WordNetLemmatizer() algorithm.
6. tokens = word_tokenize(text_data.lower())# Tokenizing the input text.

7. # Performing lemmatization with lem.lemmatize(word) function.
8. print( " ".join([lem.lemmatize(word) for word in tokens]) )
```

```
# OUTPUT: i am in america , work a a data scientist and urgent to reporting @ 10:00 am every day… !
```

By Using SpaCy:

```
1. import spacy

2. nlp = spacy.load("en_core_web_sm") # Loading the nlp model

3. # Creating document from nlp model
4. stop_words_text = nlp('i am in america, working as a data scientist
   and urgent to reporting @10:00 am every day…!')

5. # Performing lemmatization with lemma_
6. print(" ".join([token.lemma_ for token in stop_words_text]))
```

```
# OUTPUT: I be in america , work as a data scientist and urgent to report @10:00 be every day … !
```

4.5. Removing Special and Hidden Characters

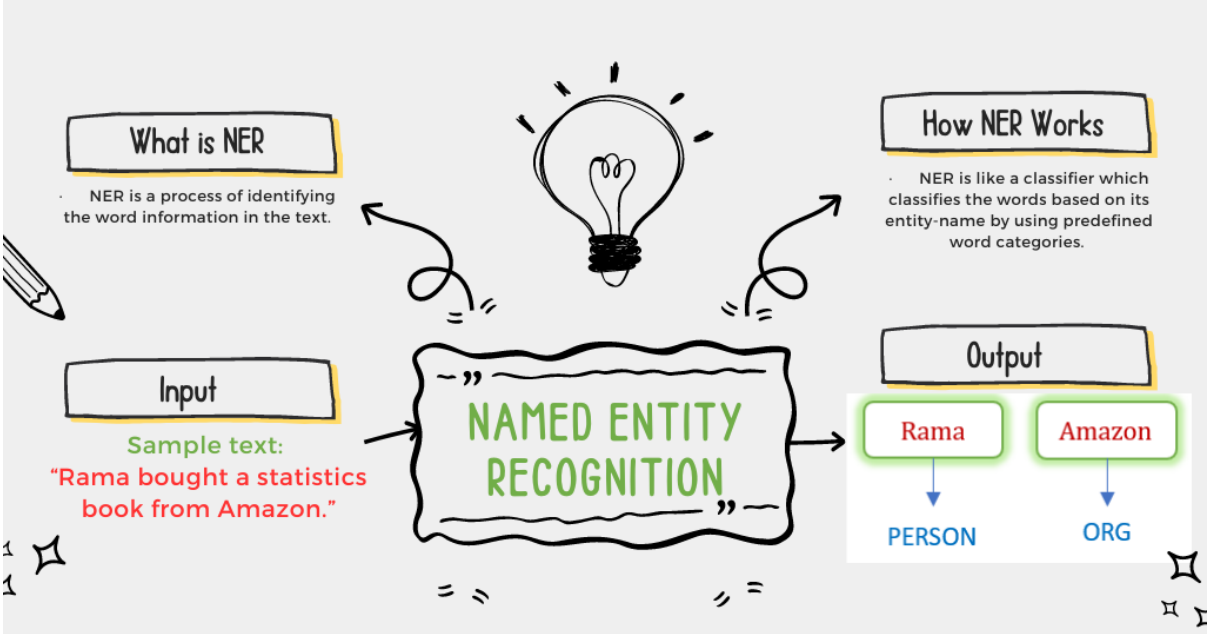
Sometimes the special characters also included in the text. But special characters are not needed in NLP, so those special/hidden characters should be removed in data cleaning stage.

Special characters: “@”, “#”, “&”, “…!” …. etc.
Special characters: “/n”, “/t”, “/s” …. etc.

```
1. import nltk
2. import re

3. def remove_special_characters(text):
```

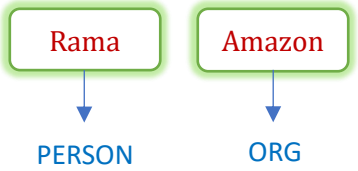
```
    words = nltk.word_tokenize(text) # Tokenize the text into words

    # Define a regular expression pattern to filter out special characters
    pattern = r"[^a-zA-Z0-9:]"

    # Use list comprehension to filter words
    return " ".join([re.sub(pattern, '', word) for word in words])

text = 'I am in America, Working as a DATA SCIENTIST and reporting @ 10:00 am every day…!'
print( remove_special_characters(text))

# OUTPUT: I am in America  Working as a DATA SCIENTIST and reporting  10:00 am every day
```

Named Entity Recognition



- Named entity recognition (NER) is a most important data preprocessing step in NLP.
- NER is a process of identifying the word information in the text.
- NER is like a classifier which classifies the words based on its entity-name by using predefined word categories.

Now let's understand how these name entities exist in a sample text:



Sample text: "Rama bought a statistics book from Amazon."

- There are many types of named entities are available in Spacy and NLTK libraries, but most General and commonly used named entities are mentioned below.

SpaCy recognizes the following built-in entity types:

PERSON	People, including fictional.
NORP	Nationalities or religious or political groups.
FAC	Buildings, airports, highways, bridges, etc.
ORG	Companies, agencies, institutions, etc.
GPE	Countries, cities, states.
LOC	Non-GPE locations, mountain ranges, bodies of water.
PRODUCT	Objects, vehicles, foods, etc. (Not services.)
EVENT	Named hurricanes, battles, wars, sports events, etc.
WORK_OF_ART	Titles of books, songs, etc.
LAW	Named documents made into laws.

LANGUAGE	Any named language.
DATE	Absolute or relative dates or periods.
TIME	Times smaller than a day.
PERCENT	Percentage, including "%".
MONEY	Monetary values, including unit.
QUANTITY	Measurements, as of weight or distance.
ORDINAL	"first", "second", etc.
CARDINAL	Numerals that do not fall under another type.

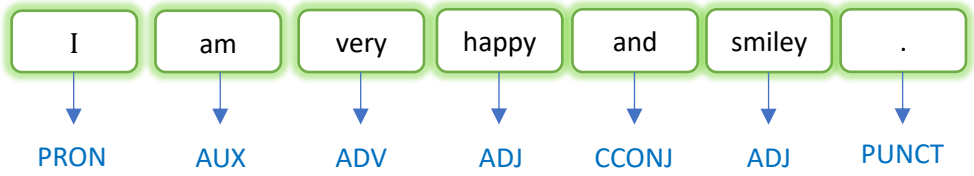
- Along with these, we will train SpaCy NER to recognize drug names as new entity. We will train 10000 reviews with drug names as DRUG entity.

Parts Of Speech Tagging

- POS tagging is the process of categorizes words from the text corpus with a particular part of speech tag.
- The POS tags represent different parts of speech, such as nouns, verbs, adjectives, adverbs, pronouns, etc.
- The parts of speech structure of a text is very useful to understand the input text structure which is using in various NLP algorithms.

Let's understand how to perform POS tagging on a sample text:

Sample text: "I am very happy and smiley."



- You can see the parts of speech tagging process in the above diagram. First we are taking a sample text sentence next each word assigned with a respective parts of speech tag.

All standard POS tags can see in the below table which are using in NLP:

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-3 sg pres	eat
JJS	Adj., superlative	wildest	VBZ	Verb, 3 sg 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	

Tag	Description	Example	Tag	Description	Example
RBR	Adverb, comparative	faster	.	Sentence-final punc	(. ! ?)
RBS	Adverb, superlative	fastest	:	Mid-sentence punc	(: ; ...)
RP	Particle	up, off			

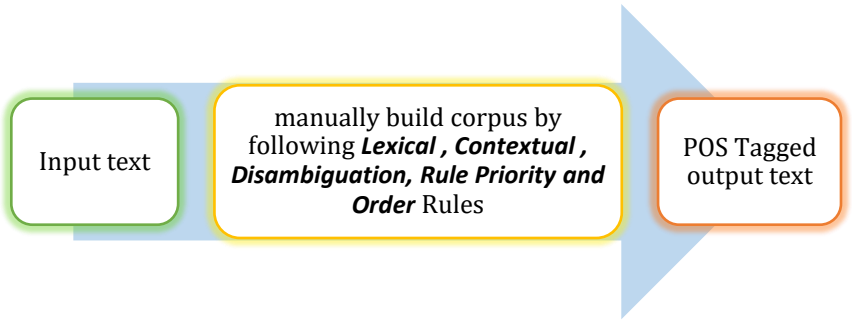
- The main important point in POS tagging is, one word can have multiple parts of speech tags based on the situation. For example, a book can be a verb (book that flight) or a noun (hand me that book).
- So, giving the perfect POS tag to a word is the main important task. If the tag assigned wrongly then the accuracy at end stage will be bad. There are some algorithms are using for POS stacking we'll discuss one by one in this chapter in below sections.

There are mainly four important POS stagging algorithm techniques are using in the NLP:

1. Rule-Based Tagging
2. Stochastic POS Tagging
3. Transformation-Based Learning Tagger: TBL
4. Hidden Markov Model POS Tagging: HMM

1. Rule-Based Tagging

Rule-based POS tagging in NLP (Natural Language Processing) involves assigning parts of speech (POS) tags to words in a text based on predefined grammatical rules.



Steps involved in rule-based POS tagging:

1. ***Tokenization:*** The input text is first tokenized into individual words or tokens.
2. ***POS Tag Dictionary:*** A rule-based POS tagger typically uses a pre-defined POS tag dictionary which is built manually. This dictionary contains a list of words and their associated POS tags. These words and POS tags built followed by below 3,4,5,6 rules.
3. ***Lexical Rules:*** Lexical Rules are a set of straightforward rules based on the spelling or appearance of a word. These rules help us determine the Part-of-Speech (POS) of a word without considering the context or surrounding words. For example, some simple lexical rules might be:
   - If a word ends with "-ing," it is likely a verb (e.g., "running," "eating").
   - If a word ends with "-ly," it is likely an adverb (e.g., "quickly," "happily").
   - If a word starts with a capital letter, it is likely a proper noun (e.g., "John," "London").
4. ***Contextual Rules:*** In more sophisticated rule-based taggers, contextual rules are applied to determine the POS tag based on the surrounding words or the grammar of the sentence. For example, if a word ends with "-ing" and is preceded by a form of "to be" (e.g., "is running," "are walking"), it is likely a verb in the present participle form.
5. ***Disambiguation:*** Rule-based POS taggers might employ additional rules or heuristics to disambiguate between different possible POS tags for a word that has multiple interpretations. For instance, the word "wind" could be a noun (referring to air in motion)

or a verb (to twist or turn). The context and neighboring words may help resolve this ambiguity.

6. ***Rule Priority and Order:*** Rule-based POS tagging systems often have a priority order for rules, and they are applied sequentially. For instance, more specific rules might be applied before general ones, or certain rules may take precedence based on linguistic considerations.

2. Stochastic POS Tagging

Stochastic POS Tagger uses probabilistic and statistical information from the corpus of POS tagged text to assign a POS tag to each word in a new sentence.

This method mainly uses two types of techniques those are:

1. Word frequency measurements
2. Tag sequence probabilities

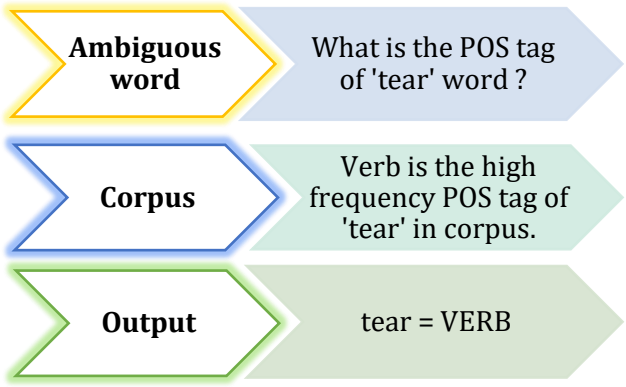
2.1. Word Frequency Measurements

This method assigns POS tag to ambiguous words based on frequency of that particular word-tag in the corpus.

Let's understand this approach using some example sentences:

Sentence 1:	"She shed a tear."	POS tag of tear is: ('tear', 'NN')
Sentence 2:	"Be careful not to tear the paper."	POS tag of tear is: ('tear', 'VERB')

The word frequency method will now check the most frequently used POS tag for "tear". Let's say this frequent POS tag happens to be VERB; then it assigns the POS tag of "tear" = VERB.



The main drawback of this approach is that it can yield invalid sequences of tags.

2.2. Tag Sequence Probabilities

In this method, the best tag for a given ambiguous word is determined by the probability that it occurs with "n" previous tags.

Simply assume a new sequence of 4 words, w1, w2, w3, w4. And we need to identify the POS tag of w4. If n = 3, we will consider the POS tags of 3 words (w1, w2, w3) prior to w4 in the labeled corpus of text. Let's say the POS tags for first 3 words are:

W1 = NOUN W2 = VERB W3 = DETERMINER

In short, N, V, D: NVD

Then in the labeled corpus of text, we will search for this NVD sequence. Let's say we found 100 such NVD sequences. Out of these, 10 sequences have the POS of the next word is NOUN, 90 sequences have the POS of the next word is VERB. Then the POS of the word W4 = VERB.

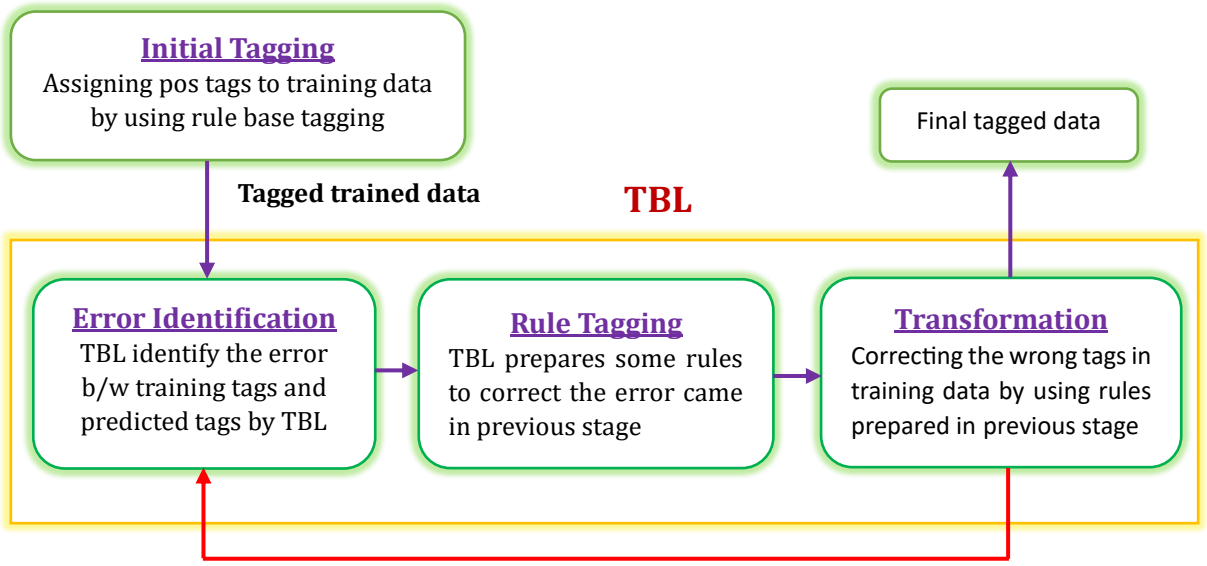
Drawbacks:

Now let's discuss some properties and limitations of the Stochastic tagging approach:

- sometimes the predicted sequence is not grammatically correct.
- This POS tagging is based on the probability of the tag occurring (either solo or in sequence)
- It requires labeled corpus, also called training data in the Machine Learning lingo
- There would be no probability for the words that don't exist in the training data
- It uses a different testing corpus (unseen text) other than the training corpus
- It is the simplest POS tagging because it chooses the most frequent tags associated with a word in the training corpus.

2. Transformation-Based Learning Tagger: TBL

- Transformation-based tagging is the combination of Rule-based & stochastic tagging methodologies.
- TBL is based on the idea of transforming an initial tagging of the training data through a series of rule-based transformations to gradually improve the accuracy of the tagger.



Here's how the TBL process works:

1. ***Initial Tagging:*** The first step is to assign initial part-of-speech tags to each word in the training data. This can be done using a simple rule-based tagger, a dictionary lookup, or any other tagging method.

2. ***Error Identification:*** After the initial tagging, TBL compares the predicted tags to the correct tags in the training data and identifies the words where the initial tagger made mistakes.
3. ***Rule Generation:*** For each identified error, TBL creates a transformation rule. A transformation rule consists of a context pattern (a sequence of words and their tags) and a target tag. The context pattern describes the local context of the word where the error occurred, and the target tag is the correct tag for that word.
4. ***Transformation:*** The transformation rules generated in the previous step are applied to the training data. Whenever the context pattern of a rule matches a portion of the training data, the target tag of that rule is applied to the word, potentially correcting the initial tag.
5. ***Iteration:*** Steps 2 to 4 are repeated for a fixed number of iterations or until a convergence criterion is met. After each iteration, the error identification process is performed again on the updated training data, generating new rules to further improve the tagger's performance.
6. ***Tagging:*** Once the TBL algorithm completes its iterations, the final transformation rules are used to tag new, unseen sentences.

TBL is an iterative process, and each iteration aims to reduce the number of errors in the training data by creating and applying transformation rules. The algorithm continues to refine its predictions until it reaches a satisfactory level of accuracy.

4. Hidden Markov Model POS Tagging: HMM

A Markov chain is useful to compute a probability for a sequence of events. Here sequence of events is nothing but sequence of POS tags for a sentence.

- Before start HMM, we should understand two terms called ***observe events*** and ***hidden events.***
- In our case ***observe events*** is nothing but words and ***hidden events*** is nothing but respective POS tags.
- The hidden Markov model is a probabilistic model, it helps to Identify sequence of hidden events (POS tags) by using both observe events (Words) and hidden events (POS tags).

4.1. Working of HMM

Step-1: First find out the probabilities of possible tag sequences for given word sequences. i.e., $P\ (T/W)$

Step-2: And select the tag sequence that has the highest probability. Mathematically we write it like: $argmax_T\ P\ (T/W)$

Word Embedding

Def: Word embedding refers to a numerical representation of words in a continuous vector space. Simply converting text into numbers (vectors) is called word embedding.

The word vector always has two types of dimensions:

Matrix dimension	This dimension indicated by number of square braces []. The number of braces indicates the dimension of the matrix.
Vector space dimension	This dimension indicated by number of values in a vector. The number of values indicates the dimension of the word in vector space.

1. Matrix Dimension:

- This refers to the number of rows and columns in a matrix. For example, a matrix with 3 rows and 4 columns is a 3x4 matrix.
- In Python-like notation (such as in NumPy arrays), the dimensions of an array are indicated by the number of nested square brackets. For instance, a 2D matrix might look like [[...], [...]] (2 sets of brackets), while a 3D tensor might look like [[[...], [...]], [[...], [...]]] (3 sets of brackets).
- The dimensions of a matrix are indicated by its number of rows and columns. In programming notation, the number of nested square brackets can show the depth or rank of the matrix, but the actual dimensions are given by the shape (e.g., 3x4 matrix).

2. Vector Space Dimension:

- This refers to the number of values (components) in a vector. For instance, if a word vector has 100 values, it is in a 100-dimensional vector space.
- This is not indicated by the number of square brackets but by the length of the vector. For example, a 3-dimensional vector might be [x, y, z].
- This dimension is indicated by the number of values in the vector. A vector with n values is an n-dimensional vector in the vector space.

3. Meaning of Each Element in Word Vector:

***Each element** in a word vector **represents** a **coordinate or a feature** of the **word** in the vector space.*
*These coordinates collectively capture **the semantic meaning of the word** in a high-dimensional space.*

Here's a more detailed explanation:

1. **Coordinate in the Vector Space:**

   Each element represents the word's position from each coordinate in vector space. For example, in a 100-dimensional vector, there are 100 coordinates.

2. **Semantic Feature:**

   These coordinates correspond to latent features that the embedding algorithm (such as Word2Vec, GloVe, or fastText) has learned from the data. Although the exact meaning of each dimension is not explicitly known, the overall vector encapsulates semantic relationships.

3. **Contextual Relationships:**

   In practice, similar words (in terms of meaning or usage) tend to have similar vectors. For instance, words like "king" and "queen" or "apple" and "orange" might be close to each other in the vector space.

4. **Magnitude and Direction:**

   The magnitude of each element affects the vector's overall direction and length in the vector space. The direction is crucial because it defines how words are related in terms of their meaning or context.

5. **Here's an illustrative example:**

   Consider a simplified 3-dimensional word vectors for words "cat" and "dog":

   $cat = [0.25, -0.10, 0.75]$

   $dog = [0.20, -0.15, 0.80]$

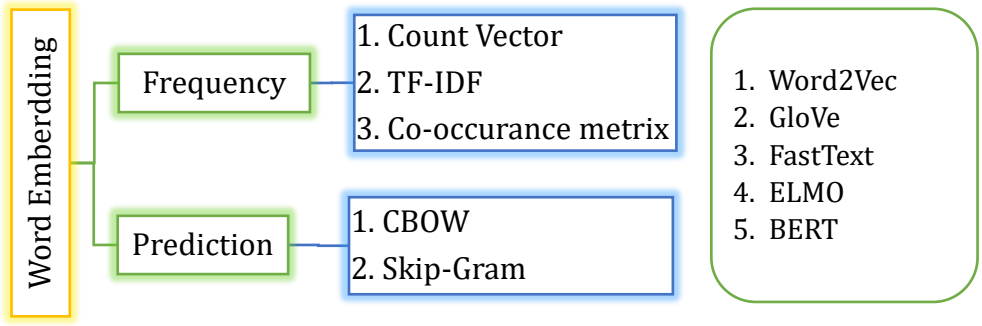
   - The first value (0.25) represents the word "cat's" coordinate along the first dimension.
   - The second value (-0.10) represents its coordinate along the second dimension.
   - The third value (0.75) represents its coordinate along the third dimension.

   We can observe that "cat" and "dog" might be close to each other in this 3D space, indicating that they are semantically related (both are animals, pets, etc.).

1. Why Word Embedding:

- Computer/ML algorithms can understand only numbers.
- A vector representation of a word is a one row array like [2.5, 5, -0.2,1, 0, -3.1].
- It is capable of capturing context of a word in a document, semantic and syntactic similarity, relation with other words, etc
- Word embedding gives similar representation where words that have the same meaning.

There are two categories of word embedding techniques:



Frequency based word embedding works based on the frequency of the word in text corpus.

2. Count Vectors:

- Count Vector working based on the frequency of each word in all documents.

2.1. How Count Vector Works:

Step-1: Taking input text:

Let's assume that our corpus has 2 documents:
Doc1 = *'He is a boy. And he is a data scientist'*
Doc2 = *'Rama is a software employee'*

Step-2: Remove stop-words and convert whole text into lower case:

Frequency of stop-words don't have impact on document, so remove stop-words before creating the count vector.

Documents After remove the stop-words:

Doc1 = *'boy. data scientist'*
Doc2 = *'Rama software employee'*

Step-3: Create words dictionary:

Create Dictionary of the corpus (unique word in the corpus):
[*'boy', 'data', 'employee', 'rama', 'scientist', 'software',*]
Here number of unique word (word corpus length) = 6.

Step-4: Create a count matrix:

	boy	*data*	*employee*	*rama*	*Scientist*	software
Doc1	0	1	1	1	0	0
Doc2	1	0	0	0	1	1

Now, a column is representing a word vector for the corresponding word.

- ✓ word 'rama' has count vector [0,1]
- ✓ Example: 'boy' has count vector [1,0]

Step-5: Prediction:

In prediction if we give a new document to the model, it will generate an array which contains 6 elements because our word corpus length is 6.

New doc: 'rama is a software employee and rama data scientist'
Output doc vector: [[0 1 1 2 1 1]]

Code:

```
1. from sklearn.feature_extraction.text import CountVectorizer

2. text = [doc1, doc2] # list of text documents

3. vectorizer = CountVectorizer()# Tokenize and build vocab
4. vectorizer.fit(text)
5. print(vectorizer.vocabulary_)

6. # Encode document
7. vector = vectorizer.transform(['rama is a software employee and rama data
   scientist'])
8. # Summarize encoded vector
9. print(vector.shape)
10. print(vector.toarray())

# OUTPUT: {'boy': 0, 'data': 1, 'scientist': 4, 'rama': 3, 'software': 5, 'employee': 2}
          (1, 6)
          [[0 1 1 2 1 1]]
```

2.2. Variation in Count Vector:

- You can put count 1 when count of any word >1, i.e., that shows that particular word is present or not.
- For very big corpus it is hard to build such doc-term matrix, so take that word whose count is greater than some threshold.

3. TF-IDF:

Def: Converting the text/words into vectors by multiplying the *Term Frequency* and *Inverse Document Frequency* is called the *TF-IDF*.

- The limitation of count vector is that common words appears frequently and frequency of important word is very less.
- TF-IDF gives the weightage to that word which occurs very less time. i.e, it captures the uniqueness.

3.1. Abbreviation of TF-IDF:

TF = Term Frequency
IDF = Inverse Document Frequency

3.2. TF-IDF Formula:

$$TF = \frac{Number\ of\ repetations\ of\ word\ in\ doc}{Number\ of\ words\ in\ doc}$$

$$IDF = \log\left[\frac{Number\ of\ docs}{Number\ of\ docs\ containing\ word}\right]$$

TF-IDF= $TF \times IDF$

3.3. Working Steps:

Step-1: Taking input text:

Let's take 3 input documents:
Doc1 = *'best book'*
Doc2 = *'best collage'*
Doc3 = *'best collage book'*

Step-2: Calculate TF for each word:

Let's calculate TF values for each word in each document by using above formula:

	best	book	collage
Doc1	1/2	1/2	0
Doc2	1/2	0	1/2
Doc3	1/3	1/3	1/3

	best	book	collage
Doc1	0.5	0.5	0
Doc2	0.5	0	0.5
Doc3	0.33	0.33	0.33

Step-3: Calculate IDF for each word:

Let's calculate IDF values for each word in each document by using above formula:

words	
best	$\log\left[\frac{3}{3}\right] = 0$
book	$\log\left[\frac{3}{2}\right] = 0.1761$
collage	$\log\left[\frac{3}{2}\right] = 0.1761$

Step-3: Calculate TF- IDF for each word:

Let's calculate TF – IDF:

	best	book	collage
Doc1	0.5	0.5	0
Doc2	0.5	0	0.5
Doc3	0.33	0.33	0.33

words	IDF
best	0
book	0.1761
collage	0.1761

	best	book	collage
Doc1	$0.5 \times 0 = 0$	$0.5 \times 0.1761 = 0.0881$	$0 \times 0.1761 = 0$
Doc2	$0.5 \times 0 = 0$	$0 \times 0.1761 = 0$	$0.5 \times 0.1761 = 0.0881$
Doc3	$0.33 \times 0 = 0$	$0.33 \times 0.1761 = 0.0581$	$0.33 \times 0.176 = 0.0581$

3.4. Variation of TF-IDF:

- TF-IDF is the multiplication of TF and IDF, if TF value is very high then TF-IDF value will be more. To avoid this, we use logarithmic method called BM25.

4. Co-Occurrence Matrix:

Def: Co-Occurrence Matrix is the process of counting the occurrence of a word with its neighbor word. Will understand this definition perfectly in blow working concept.

4.1. Main Purpose:

- Similar words tend to occur together and will have similar context for example. *Tiger is an animal*, *Lion is an animal*.
- Apple and mango tend to have a similar context i.e fruit.

4.2. What is Window:

Before start learning about co-occurrence matrix, we should understand one topic called window in text data.

- The window is nothing but n-gram. We can select any type of gram (like bigram trigram) and this gram number is called window.
- The window is a number and it indicates the direction of the context.
- If we select bigram as window then window number is 2. Like that 3 for trigram and so on.
- The minimum size of window should be 1 and it is default value.
- Once the window selected, then calculation of co-occurrence performed between the windows not between words.
  For Ex: If we select window is 2 for input text *"Tiger is an animal".*

Tiger is an animal	***Tiger is*** occurred with ***an animal***
Tiger is an animal	***Is an*** occurred with ***Tiger,*** and ***animal***
Tiger is an animal	***an animal*** occurred with ***Tiger is***

- In above table highlighted part is the window.

4.3. Working:

Step-1: Taking input text:

Let's take 2 input documents:

Doc1 = *'Tiger is an animal and it eats meat'*

Doc2 = *'Lion is an animal and it eats meat'*

After removing stop-words:

Doc1 = *'Tiger animal eat meat'*

Doc2 = *'Lion animal eat meat'*

Step-2: Selecting the window:

Let's select window values is 1 for one example and 2 for another example.

Step-3: Calculating co-occurrence matrix:

Window -1

	animal	eat	lion	meat	tiger
animal	0	2	1	2	1
eat	2	0	1	2	1
lion	1	1	0	1	0
meat	2	2	1	0	1
tiger	1	1	0	1	0

Window - 2

	animal eat	eat meat	lion animal	tiger animal
animal eat	0	2	1	1
eat meat	2	0	1	1
lion animal	1	1	0	0
tiger animal	1	1	0	0

5. Word2Vec:

History: Word2Vec model is used for word representations in *vector space.* It is founded by ***Tomas Mikolov*** and a group of the research teams from Google in 2013.

- This embedding captures the semantic relationships between words and enable machines to understand the meaning of words in a more efficient manner.
- It is a neural-network based model and it build with CBOW and Skip-Gram techniques.

5.1. Word2Vec Architecture:

There are two main architectures for implementing word2vec:

1. Continuous Bag of Words (CBOW)
2. Skip-gram.

Both architectures aim to learn word embeddings by considering the context in which words appear.

Continuous Bag of Words (CBOW):

In the CBOW architecture, the model tries to predict a target word based on its surrounding context words. The context words are treated as input, and the target word is the output. The model is trained to minimize the difference between the predicted target word and the actual target word.

Skip-Gram:

In the Skip-gram architecture, the model does the opposite: it predicts the context words given a target word. The target word is treated as input, and the context words are the output. Similarly, the model is trained to minimize the difference between predicted and actual context words.

- The key idea behind both CBOW and Skip-gram is to learn word embeddings that encode semantic relationships between words.

- Due to that semantic relationship encode the similar words are close to each other in the embedding space.
- This is achieved through training a neural network with a large amount of text data.

CBOW

✓ CBOW - Continuous Bag of Words
✓ The goal of CBOW is to predict a target word given the context words in a sentence.

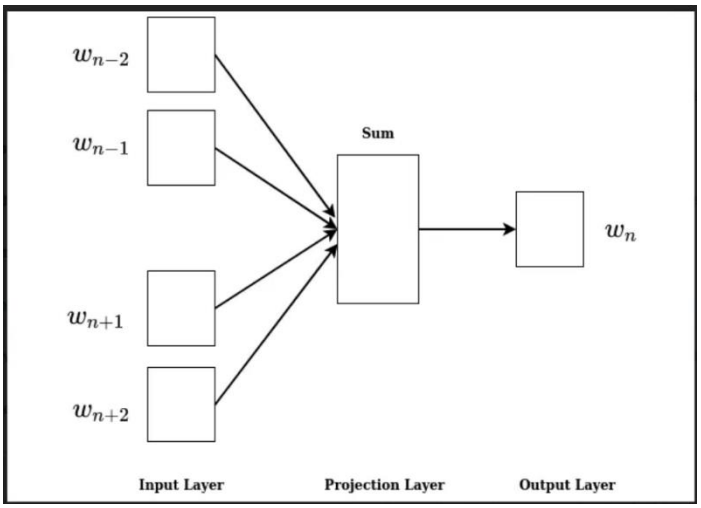
1. Used for:

- Predicting missing words or similar words.
- Code embedding.

2. Architecher:

2.1. Architecher Parts:

- The architecture of the CBOW consisting of an input layer, a hidden layer, and an output layer.
- The input layer is used to represent the context words, the hidden layer is used to learn the word embeddings, and the output layer is used to predict the target word.



2.2. Input Layer:

The input layer is typically represented by a one-hot encoded vector, where each element in the vector corresponds to a specific word in the vocabulary. For example, if the vocabulary contains 10,000 words, the input layer will have 10,000 elements.

2.3. Hidden Layer:

The hidden layer is where the word embeddings are learned. It is a dense layer, with each neuron representing a specific word in the vocabulary. The number of neurons in the hidden layer is the same as the number of words in the vocabulary.

2.4. Output Layer:

The output layer is also a dense layer, with each neuron representing a specific word in the vocabulary. The number of neurons in the output layer is the same as the number of words in the vocabulary.

3. Working:

- Let x(1), x(2), …, x(n) be the context words, where x(i) is a one-hot encoded vector.
- Let W(1) be the weight matrix connecting the input layer to the hidden layer, and W(2) be the weight matrix connecting the hidden layer to the output layer.

- Let h be the hidden layer, which is the average of the input vectors, h = 1/n * (x(1) + x(2) + … + x(n))
- Let y be the output layer, which is the probability distribution over the vocabulary, y = softmax(W(2) * h)
- The target word is selected as the word with the highest probability in y.

Note: The training process for CBOW is to minimize the difference between the predicted probability distribution y and the actual target word using a loss function such as cross-entropy loss.