NLP Complete Notes – Tauqueer Alam

UNIT - 1

Computing with Language: Texts and Words

This is one of the **first chapters** when learning NLP using Python (especially with NLTK).
It focuses on **how computers handle text**, and **how we can analyze language data** computationally.

Let's break it down

What It Means

"Computing with Language" means using **Python programs** to:

- Process large collections of text (called *corpora*),
- Count and search words,
- Analyze word patterns and frequencies,
- Understand structure and meaning in human language.

So the **goal** is to use Python to treat *language as data* and do useful computations on it.

Common NLP Tasks Here

Task	Description	Example
Tokenization	Splitting text into words or sentences	`"Hello world!" → ["Hello", "world", "!"]`
Counting Words	Finding frequency of each word	Count how many times "Python" appears
Concordance	Find occurrences of a word and its surrounding words	Find all places where "science" occurs in a text
Collocation	Commonly occurring word pairs	"Machine learning", "New York"
Dispersion Plot	Shows where words appear in the text	Plot "freedom" and "war" in a novel

A Closer Look at Python: Texts as Lists of Words

Once you have text data, you need to **represent and manipulate it**.
In Python, text can be treated as:

- **Strings** (continuous sequences of characters), or
- **Lists of words** (tokens).

Texts as Lists

If you split a text into words using `split()` or NLTK's tokenizer, you get a **list**:

```
sentence = "I love natural language processing"
words = sentence.split()
print(words)
# Output: ['I', 'love', 'natural', 'language', 'processing']
```

Now you can use **Python list operations**:

```
print(words[0])       # 'I'
print(words[-1])      # 'processing'
print(len(words))     # 5
print(sorted(set(words)))  # alphabetically sorted unique words
```

Why Treat Text as a List?

Because:

- You can **loop**, **count**, **slice**, and **search** words easily.
- It helps in **feature extraction**, **frequency distribution**, and **pattern matching**.

Example: Word Frequency

```
from nltk import FreqDist

text = "Python is great and Python is popular for NLP"
words = text.split()

fdist = FreqDist(words)
print(fdist.most_common(3))
# [('Python', 2), ('is', 2), ('great', 1)]
```

This tells you which words appear most frequently — very useful in text analysis.

Common Python List Operations for NLP

Operation	Example	Result
Indexing	`words[1]`	2nd word
Slicing	`words[1:3]`	subset of words
Membership	`'Python' in words`	`True`
Concatenation	`words + ['rocks!']`	add new words
Iteration	`for w in words:`	loop through text

Example Combined

```
import nltk
from nltk import FreqDist
from nltk.tokenize import word_tokenize

sentence = "NLTK makes natural language processing simple and powerful."
tokens = word_tokenize(sentence)

print("Tokens:", tokens)
print("First word:", tokens[0])
print("Word frequency:", FreqDist(tokens))
```

Computing with Language: Simple Statistics

This topic introduces **basic statistical analysis on text data** — one of the most important foundations for NLP and Data Science.

What It Means

You learn how to use **mathematics and statistics** to extract useful information from language — like **word frequency**, **richness of vocabulary**, or **word distributions**.

It's about quantifying **how language behaves**.

Common Statistical Measures in NLP

Concept	Description	Example
Frequency Distribution (FDist)	How often each word appears	"Python" appears 50 times
Lexical Diversity	Ratio of unique words to total words	`len(set(words)) / len(words)`
Word Length Distribution	Average or histogram of word lengths	Mean word length = 5.2
Conditional Frequency	Frequency of words under certain conditions	How often "news" occurs after "fake"

Example in Python (Using NLTK

```
from nltk import FreqDist
from nltk.corpus import gutenberg

# Load a text
words = gutenberg.words('austen-emma.txt')

# Frequency distribution
fdist = FreqDist(words)

print("Most common words:", fdist.most_common(5))
print("Lexical diversity:", len(set(words)) / len(words))
```

OUTPUT

```
Most common words: [('the', 5200), ('to', 4200), ('and', 4000), ...]
Lexical diversity: 0.07
```

Why Important?

Simple statistics give us:

- Insights about text structure (how repetitive or rich it is)
- Data for **feature engineering** in ML models
- Basis for **topic modeling** or **document comparison**

Back to Python: Making Decisions and Taking Control

Now we switch back to **Python concepts** that help control program flow — essential for **building NLP pipelines** that make decisions automatically.

What It Means

Here you learn how to use:

- **Conditional statements** (`if`, `elif`, `else`)
- **Loops** (`for`, `while`)
- **Functions** (`def`)
- **Comprehensions** (like `[w for w in words if len(w) > 5]`)

These let your program **make decisions**, **filter data**, and **react** to text patterns dynamically.

Example 1 — Using Conditions

```
word = "Python"

if word.endswith("on"):
    print("Ends with 'on'")
else:
    print("Does not end with 'on'")
```

Output: Ends with 'on'

Example 2 — Using Loops in Text Processing

```
sentence = "I love learning Natural Language Processing"
words = sentence.split()

for w in words:
    if len(w) > 5:
        print(w)
```

OUTPUT:

learning

Natural

Language

Processing

Why Important?

Because NLP programs need to:

- **Filter** specific kinds of words (e.g., nouns, verbs, stopwords)
- **Handle multiple conditions** (e.g., if token is alphabetic, not numeric)
- **Control flow** (e.g., skip punctuation, lowercase all words, etc.)

So this part ensures you can *control text analysis intelligently*.

Automatic Natural Language Understanding

This is where we shift from *counting and manipulating words* → to *understanding meaning*.

It introduces **the goal of NLP** — enabling computers to *understand* and *respond* to human language automatically.

What It Means

Automatic Natural Language Understanding (NLU) is the ability of a computer to:

- **Interpret** human language (text or speech)
- **Extract meaning** (semantics, intent, entities)
- **Generate responses** intelligently

Subfields Involved

Area	Description	Example
Tokenization	Breaking text into words/sentences	"I love NLP" → ["I", "love", "NLP"]
POS Tagging	Identifying part of speech	"love" → verb
Named Entity Recognition (NER)	Identifying names, places, dates	"Elon Musk" → PERSON
Parsing	Analyzing sentence structure	Grammar trees
Semantic Analysis	Understanding meaning of	"bank" → riverbank or

Area	Description	Example
	text	financial bank
Sentiment Analysis	Detecting opinion or emotion	“good” → positive
Coreference Resolution	Linking pronouns to nouns	“He” → “John”
Machine Translation	Converting languages	English → Hindi

Example Using NLTK

```
import nltk
from nltk import pos_tag, word_tokenize, ne_chunk

sentence = "Elon Musk founded SpaceX in 2002."

tokens = word_tokenize(sentence)
tags = pos_tag(tokens)
entities = ne_chunk(tags)

print("Tokens:", tokens)
print("POS Tags:", tags)
print("Named Entities:", entities)
```

OUTPUT

```
Tokens: ['Elon', 'Musk', 'founded', 'SpaceX', 'in', '2002', '.']
POS Tags: [('Elon', 'NNP'), ('Musk', 'NNP'), ('founded', 'VBD'), ...]
Named Entities:
(S
  (PERSON Elon/NNP)
  (PERSON Musk/NNP)
  founded/VBD
  (ORGANIZATION SpaceX/NNP)
  in/IN
  2002/CD
  ./.)
```

Why Important?

Because NLU is what enables:

- Chatbots (like Siri, Alexa, ChatGPT □)
- Sentiment analysis
- Search engines
- Translation systems
- Question-answering bots
- Voice assistants

It's the *"intelligent"* side of NLP.

Accessing Text Corpora

What Is a Corpus?

A **corpus** (plural: *corpora*) is a **large collection of text** — like books, news articles, tweets, or speech transcripts — used for language research and NLP model training.

In NLP, corpora are used to:

- Analyze language structure
- Train models (for tagging, translation, sentiment, etc.)
- Study word usage and frequency

Accessing Corpora in NLTK

NLTK provides many built-in corpora.

```
from nltk.corpus import gutenberg, brown, reuters

# List available files
print(gutenberg.fileids())
print(brown.categories())
print(reuters.categories())
```

Example: Reading Text

```
from nltk.corpus import gutenberg

# Access Jane Austen’s 'Emma'
words = gutenberg.words('austen-emma.txt')
print("Total words:", len(words))
print("First 20 words:", words[:20])
```

OUTPUT

```
Total words: 192427
First 20 words: ['[', 'Emma', 'by', 'Jane', 'Austen', ...]
```

Corpus Operations

Operation	Description	Example
`.words()`	Returns list of all words	`gutenberg.words('austen-emma.txt')`
`.sents()`	Returns list of sentences (each sentence = list of words)	`brown.sents(categories='news')`
`.raw()`	Returns entire text as one string	`gutenberg.raw('austen-emma.txt')`

Why Important?

Accessing corpora lets you:

- Work with *real-world text*
- Compute statistics (word count, frequency, diversity)
- Train and evaluate models on large text data

What is a Conditional Frequency Distribution?

A **Conditional Frequency Distribution** (CFD) in NLP is used to find **how often something happens under certain conditions**.

Think of it like:

"How many times does a word appear in a specific category (condition)?"

Example to Understand

Imagine you have two categories (conditions):

- **News**
- **Romance**

Each category has words (data).

Category	**Words**
News	"war", "president", "election", "war", "budget"
Romance	"love", "kiss", "love", "heart", "beautiful"

Now you want to know:

- How many times the word "love" appears in romance?
- How many times the word "war" appears in news?

Example in Python (Using NLTK)

```
from nltk import ConditionalFreqDist

# Create sample data (pairs of condition and word)
data = [
    ('news', 'war'),
    ('news', 'president'),
    ('news', 'war'),
    ('romance', 'love'),
    ('romance', 'kiss'),
    ('romance', 'love')
]

# Create Conditional Frequency Distribution
cfd = ConditionalFreqDist(data)

# Print how often each word appears in each condition
print(cfd['news'].items())      # Output: dict_items([('war', 2), ('president', 1)])
print(cfd['romance'].items())   # Output: dict_items([('love', 2), ('kiss', 1)])

# Find specific counts
print("'war' in news:", cfd['news']['war'])
print("'love' in romance:", cfd['romance']['love'])
```

OUTPUT

```
'war' in news: 2
'love' in romance: 2
```

Why Useful?

It helps analyze *word usage patterns*:

- Compare words across genres or time periods
- Understand context-based frequency
- Build features for text classification

Lexical Resources

What Are Lexical Resources?

These are **structured databases of words** — collections that tell you:

- Meanings
- Synonyms / antonyms
- Parts of speech
- Example usage

Examples include:

- **WordNet** (most popular in NLP)
- Stopwords lists
- Pronunciation dictionaries
- Sentiment lexicons

Example: Stopwords

Stopwords are **common words** like *is, the, a, in* — usually removed before analysis.

```
from nltk.corpus import stopwords

print(stopwords.words('english')[:10])
```

OUTPUT

['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're"]

Why Important?

Lexical resources give **semantic and linguistic structure** — essential for:

- Lemmatization (getting word roots)
- Synonym/antonym detection
- Sentiment or tone detection
- Building knowledge-based systems

WordNet

What Is WordNet?

WordNet is a large lexical database of English.
It groups English words into **synsets** (sets of synonyms) and records relationships between them — like:

- Synonyms
- Antonyms
- Hypernyms (is-a)
- Hyponyms (sub-type)
- Meronyms (part-of)

Why WordNet Is Important

WordNet is crucial in NLP for:

- **Semantic analysis** (understanding meaning)
- **Text classification** using word relations
- **Word sense disambiguation**
- **Question answering and summarization**
- **Knowledge graphs and ontology-based AI**

UNIT – 2

Processing Raw Text – Accessing Text from the Web and from Disk

This topic teaches how to **get raw text data** (like articles, books, or tweets) into Python for NLP tasks.
Before we analyze or clean text, we must **access (load)** it — either from the **internet (web)** or from our **computer (disk)**

Accessing Text from the Web

In NLP, we often need text from online sources — like web pages, blogs, or Wikipedia articles.

Common Ways to Access Text from the Web

(a) Using `urllib` (Built-in Python Library)

`urllib` lets us open URLs and read the text (HTML content) of web pages.

```
from urllib import request

url = "https://www.gutenberg.org/files/2554/2554-0.txt"  # Example: Crime and Punishment
response = request.urlopen(url)
raw = response.read().decode('utf8')  # Decode bytes to text

print(type(raw))
print(len(raw))
print(raw[:500])  # print first 500 characters
```

Explanation:

- `urlopen()` → opens the web page
- `read()` → reads the content
- `decode('utf8')` → converts it into a readable string

(b) Using `requests` library (simpler & modern)

```
import requests

url = "https://www.gutenberg.org/files/84/84-0.txt"  # Frankenstein
response = requests.get(url)
text = response.text

print(text[:300])
```

`requests` is easier and cleaner than `urllib`.

(c) Removing HTML Tags (if web page has HTML)

Web pages often contain tags like `<p>` or `<div>`.
We can remove them using **BeautifulSoup** (a web-scraping library).

```
from bs4 import BeautifulSoup
import requests

url = "https://www.gutenberg.org/files/11/11-h/11-h.htm"  # Alice in Wonderland
html = requests.get(url).text
soup = BeautifulSoup(html, "html.parser")
text = soup.get_text()  # extract only visible text
print(text[:500])
```

Now you have **pure text** (no HTML)

Accessing Text from Disk (Local Files)

If the text is already stored on your computer (like `.txt`, `.csv`, `.docx`), you can read it easily in Python.

(a) Reading a Text File

```
# Open and read text file
file = open("sample.txt", "r", encoding="utf8")
text = file.read()
file.close()

print(text[:200])
```

r" means read mode.
Always close the file after reading.

(b) Using with (Recommended)

```
with open("sample.txt", "r", encoding="utf8") as f:
    text = f.read()
print(text[:200])
```

Automatically closes file — safer and cleaner.

(c) Reading Multiple Files

If you have many text files in a folder:

```python
import os

folder_path = "C:/Users/Tauqueer/Documents/NLP_Texts"

for file_name in os.listdir(folder_path):
    if file_name.endswith(".txt"):
        with open(os.path.join(folder_path, file_name), 'r', encoding='utf8') as f:
            text = f.read()
            print(f"File: {file_name}, Length: {len(text)}")
```

Processing the Text

After loading text (from web or disk), you usually want to:

1. **Tokenize** → split text into words or sentences
2. **Normalize** → lowercase, remove punctuation, etc.

Example:

```python
import nltk
from nltk import word_tokenize

tokens = word_tokenize(text)
print(tokens[:20])
```

Strings — Text Processing at the Lowest Level

In NLP, everything starts with **text**, and in Python, **text = string**.
Before we use advanced tools (like NLTK tokenizers), we should understand **how strings work**, because they are the **lowest-level representation of text** in Python.

What is a String?

A **string** is a sequence of characters — letters, numbers, symbols, or spaces — enclosed in quotes.

```
text = "Natural Language Processing with Python"
print(text)
```

Accessing Characters in a String

You can access any character by its **index number** (just like list indexing).
Indexing starts from **0**.

```
text = "Python"
print(text[0])   # P
print(text[3])   # h
print(text[-1])  # n (last character)
```

String Slicing

You can extract parts of strings using **slice notation** `[start:end]`.

```
text = "Language"
print(text[0:4])   # Lang
print(text[2:])    # nguage
print(text[:5])    # Langu
```

String Operations

Python provides many useful string operations for text processing:

Operation	Description	Example
`+`	Concatenate strings	`"Hello " + "World"` → `"Hello World"`
`*`	Repeat string	`"Hi!" * 3` → `"Hi!Hi!Hi!"`
`len()`	Find length	`len("Python")` → `6`
`in`	Check substring	`"Lang" in "Language"` → `True`

String Methods for Text Cleaning

Method	Function	Example	Output
`lower()`	Convert to lowercase	`"PYTHON".lower()`	`python`
`upper()`	Convert to uppercase	`"python".upper()`	`PYTHON`
`title()`	Capitalize each word	`"hello world".title()`	`Hello World`
`strip()`	Remove spaces	`" text ".strip()`	`text`
`replace()`	Replace substring	`"AI is cool".replace("cool", "fun")`	`AI is fun`
`split()`	Split string into words	`"AI with Python".split()`	`['AI', 'with', 'Python']`
`join()`	Join list into string	`" ".join(['AI', 'with', 'Python'])`	`AI with Python`

Checking String Content

Function	Purpose	Example	Output
`isalpha()`	Checks if all characters are letters	`"Hello".isalpha()`	`True`
`isdigit()`	Checks if all characters are digits	`"123".isdigit()`	`True`
`isalnum()`	Checks if alphanumeric	`"AI123".isalnum()`	`True`
`isspace()`	Checks if only spaces	`" ".isspace()`	`True`

Example: Basic Text Preprocessing Using Strings

```
sentence = "   Natural Language Processing, or NLP, is AMAZING!   "

# Clean text
clean = sentence.strip().lower().replace(",", "").replace("!", "")
words = clean.split()

print(words)
```

OUTPUT

['natural', 'language', 'processing', 'or', 'nlp', 'is', 'amazing']

Text Processing with Unicode

When working with Natural Language Processing (NLP), we often deal with **many languages**, **symbols**, and **special characters**.
To process all of them correctly, Python uses a system called **Unicode** — a universal way to represent text from *every* language.

What is Unicode?

- **Unicode** is a standard that assigns a unique number (called a *code point*) to every character in every language.
- It solves the problem of earlier encodings (like ASCII) that could only handle English letters.

Character	Unicode Code Point	Description
A	U+0041	English Capital A
a	U+0061	English Small a
अ	U+0905	Hindi Letter A
中	U+4E2D	Chinese Character
□	U+1F600	Emoji: Grinning Face

Every symbol has its own unique code — making it possible to mix languages safely in the same file.

Encoding and Decoding

Encoding = converting text → bytes
Decoding = converting bytes → text

This is important when reading/writing files or transferring text across the web.

```
text = "नमस्ते"
encoded = text.encode('utf-8')   # convert to bytes
print(encoded)

decoded = encoded.decode('utf-8')  # convert back to text
print(decoded)
```

OUTPUT

```
b'\xe0\xa4\xa8\xe0\xa4\xae\xe0\xa4\xb8\xe0\xa5\x8d\xe0\xa4\xa4\xe0\xa5\x87'
नमस्ते
```

'utf-8' is the most common encoding — supports all languages.

Regular Expressions for Detecting Word Patterns

1. Introduction

- Regular expressions (also called **regex**) are powerful tools used to **find, match, and manipulate text patterns** in strings.

- In NLP, they are often used for **tokenization**, **pattern matching**, **cleaning text**, and **information extraction** (like finding emails, phone numbers, dates, etc.).

Example:
If you want to find all words starting with a capital letter in a paragraph, you can do it easily using a regular expression

Importing Regex Module in Python

Python provides the **`re`** module to work with regular expressions.

```
import re
```

Basic Regex Functions

Function	Description
`re.match()`	Checks if the pattern matches at the beginning of the string
`re.search()`	Searches for the first occurrence of the pattern
`re.findall()`	Returns all occurrences of the pattern
`re.sub()`	Replaces text that matches the pattern
`re.split()`	Splits a string using the pattern as delimiter

Common Regex Symbols

Symbol	Meaning	Example	Matches
`.`	Any character except newline	`h.t`	“hat”, “hit”, “hot”
`^`	Start of string	`^Hello`	Matches if string starts with "Hello"
`$`	End of string	`world$`	Matches if string ends with "world"
`\d`	Any digit (0–9)	`\d+`	"123", "56"
`\w`	Any word character (a–z, A–Z, 0–9, _)	`\w+`	"hello", "Python3"
`\s`	Any whitespace	`\s+`	space, tab, newline
`*`	0 or more repetitions	`ab*`	"a", "ab", "abb", "abbb"
`+`	1 or more repetitions	`ab+`	"ab", "abb"
`?`	0 or 1 occurrence	`colou?r`	"color", "colour"

Symbol	Meaning	Example	Matches
[]	Set of characters	[aeiou]	matches vowels
{m,n}	Between m and n repetitions	\d{2,4}	"12", "2024"
`	`	OR condition	`cat

Example 1: Find All Words Starting with Capital Letter

```
import re

text = "My name is Tauqueer Alam and I study Computer Science."
words = re.findall(r'\b[A-Z][a-z]*\b', text)
print(words)
```

OUTPUT

```
['My', 'Tauqueer', 'Alam', 'I', 'Computer', 'Science']
```

Example 2: Extract All Email Addresses

```
text = "You can contact us at info@company.com or support@helpdesk.org"
emails = re.findall(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b', text)
print(emails)
```

OUTPUT

```
['info@company.com', 'support@helpdesk.org']
```

Useful Applications of Regular Expressions

(a) Tokenization

Splitting sentences or paragraphs into words or tokens.

(b) Removing Unwanted Characters

Cleaning text by removing punctuation, special characters, or numbers.

(c) Extracting Email Addresses

Finding and collecting all emails from a large text (useful for scraping or contact extraction).

(d) Extracting Phone Numbers

Finding phone numbers in documents or web pages.

(e) Extracting Dates

Detecting date formats like `12/10/2025` or `2025-10-12`.

(f) Detecting Capitalized Words (e.g., Names, Locations)

Useful in **Named Entity Recognition (NER)** or for extracting proper nouns.

(g) Removing Extra Spaces

Cleaning messy text with multiple spaces or tabs.

(h) Extracting Hashtags or Mentions (for Social Media Data)

Very useful in NLP when analyzing tweets or Instagram captions.

Normalizing Text

1. Introduction

In Natural Language Processing (NLP), **text normalization** means converting text into a **standard or uniform format** so that it can be easily processed by algorithms.

Human language is very **inconsistent** — we write the same thing in different ways:

- "U" and "you" mean the same.
- "Running", "runs", and "ran" are forms of "run".
- "I'm" and "I am" are equivalent.

To make text **consistent**, we perform **normalization** before feeding it to any NLP model.

Why Normalization is Important

Because:

- It **reduces variations** in words that mean the same thing.
- It **improves accuracy** of NLP models.
- It makes text **clean, consistent, and comparable**.

Example:
Without normalization:
`["Running", "runs", "Ran"]`
After normalization:
`["run", "run", "run"]`

Common Text Normalization Techniques

(a) Lowercasing

Convert all characters to lowercase to avoid duplication.

```
text = "Natural Language Processing is FUN!"
text = text.lower()
print(text)
```

OUTPUT

```
natural language processing is fun!
```

(b) Removing Punctuation and Special Characters

Punctuation marks are usually not meaningful for NLP tasks.

```
import re
text = "Hello, World!!! It's Tauqueer :)"
clean_text = re.sub(r'[^\w\s]', '', text)
print(clean_text)
```

OUTPUT

```
Hello World Its Tauqueer
```

(c) Removing Stopwords

Stopwords are common words like “is”, “the”, “a”, “an”, etc., which do not add meaning.

```
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
import nltk
nltk.download('punkt')
nltk.download('stopwords')

text = "This is an example showing the removal of stopwords."
words = word_tokenize(text)
filtered = [word for word in words if word.lower() not in stopwords.words('english')]
print(filtered)
```

OUTPUT

```
['example', 'showing', 'removal', 'stopwords']
```

(d) Stemming

Stemming reduces words to their **root form** (not necessarily a real word).

Example:
“Playing”, “played”, “plays” → “play”

(e) Lemmatization

Lemmatization also reduces words to their base form (**lemma**), but it uses **vocabulary and grammar** for more accuracy.

Example:
“Better” → “good” (correct lemma)
“Was” → “be”

(f) Expanding Contractions

Converting shortened words to their full form:

- “I’m” → “I am”
- “don’t” → “do not”

Combined Example

Let’s apply multiple normalization steps together

```python
import re
from nltk.stem import PorterStemmer
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
import nltk
nltk.download('punkt')
nltk.download('stopwords')

text = "Running, Runs and RAN are forms of the word RUNNING!"
text = text.lower()  # lowercasing
text = re.sub(r'[^\w\s]', '', text)  # remove punctuation
words = word_tokenize(text)
words = [w for w in words if w not in stopwords.words('english')]  # remove stopwords
ps = PorterStemmer()
words = [ps.stem(w) for w in words]  # stemming
print(words)
```

OUTPUT

```
['run', 'run', 'ran', 'form', 'word', 'run']
```

Regular Expressions for Tokenizing Text

1. Introduction

Tokenization is the process of **splitting text into smaller units** — usually words, sentences, or phrases.
Regular expressions (**regex**) help define **patterns** that tell the computer **where to split the text**.

In simple terms:

Tokenization = breaking text into **tokens** (words or sentences) using **rules or patterns**.

Why Use Regular Expressions for Tokenization?

Regular expressions give **more control and flexibility** compared to simple splitting (like `split()` in Python).

- `split()` just divides on spaces.
- But regex can handle punctuation, numbers, contractions, etc.
- It's especially useful for **complex text** like social media data, reviews, or web content.

Simple Word Tokenization Using Regex

Let's split a sentence into words using a simple regex pattern.

```
import re

text = "Hello! My name is Tauqueer Alam. I study NLP."
tokens = re.findall(r'\b\w+\b', text)
print(tokens)
```

OUTPUT

```
['Hello', 'My', 'name', 'is', 'Tauqueer', 'Alam', 'I', 'study', 'NLP']
```

`\b\w+\b` means:

- `\b` → word boundary
- `\w+` → one or more word characters (letters, digits, underscore)

Segmentation in NLP

1. Introduction

In Natural Language Processing (NLP), **segmentation** refers to the process of **dividing text into meaningful units**, like:

- **Sentences** → Sentence Segmentation (or Sentence Boundary Detection)
- **Words** → Word Segmentation

Segmentation is an important **preprocessing step** because many NLP tasks (like tokenization, parsing, and machine translation) require text to be in smaller, meaningful units.

Types of Segmentation

(a) Sentence Segmentation

Dividing text into sentences.

Challenges:

- Punctuation ambiguity (e.g., `Dr. Smith is here.` → `Dr.` is not the end of a sentence)
- Abbreviations, decimal points, URLs

Example using regex:

```
import re

text = "Hello! I am Tauqueer Alam. How are you today? I hope you're fine."
sentences = re.split(r'(?<=[.!?])\s+', text)
print(sentences)
```

OUTPUT

```
['Hello!', 'I am Tauqueer Alam.', 'How are you today?', "I hope you're fine."]
```

Explanation:

- `(?<=[.!?])\s+` → split at spaces **after** a period, exclamation, or question mark.

Example using NLTK:

```
from nltk.tokenize import sent_tokenize
import nltk
nltk.download('punkt')

text = "Dr. Alam is a professor. He teaches NLP."
sentences = sent_tokenize(text)
print(sentences)
```

OUTPUT

```
['Dr. Alam is a professor.', 'He teaches NLP.']
```

(b) Word Segmentation

Dividing a sentence into **words** (or tokens).

Example using regex:

```python
import re
sentence = "Natural Language Processing is amazing!"
words = re.findall(r'\b\w+\b', sentence)
print(words)
```

OUTPUT

```
['Natural', 'Language', 'Processing', 'is', 'amazing']
```

Example using NLTK:

```python
from nltk.tokenize import word_tokenize
words = word_tokenize(sentence)
print(words)
```

OUTPUT

```
['Natural', 'Language', 'Processing', 'is', 'amazing', '!']
```

Notice that `word_tokenize` **keeps punctuation** as separate tokens.

Special Cases in Segmentation

1. **Languages without spaces**
   - Example: Chinese, Japanese
   - Words are not separated by spaces, so word segmentation requires **dictionary-based or statistical methods**.
2. **URLs, Emails, Hashtags, Emojis**
   - These require **custom tokenization rules** to avoid splitting in the middle.
3. **Abbreviations**
   - Example: `U.S.A.` → shouldn't split at every period.

Why Segmentation is Important

- It allows **accurate tokenization**
- It’s crucial for **POS tagging, parsing, and translation**
- Helps in **information retrieval** and **text analytics**

Formatting: From Lists to Strings in NLP

1. Introduction

In NLP, after tokenizing text, we often have **lists of words or tokens**.
Sometimes, we need to **convert these lists back into readable text** for output, display, or further processing. This process is called **formatting lists into strings**.

Why It’s Useful

- Joining tokenized words back into **sentences**.
- Preparing text for **storage, display, or machine learning models**.
- Combining outputs from **preprocessing steps** like tokenization, stemming, or lemmatization.

Converting Lists to Strings Using `join()`

The most common way in Python is using the **`join()`** method.

```
words = ['Natural', 'Language', 'Processing', 'is', 'fun']
sentence = ' '.join(words)
print(sentence)
```

OUTPUT

```
Natural Language Processing is fun
```

`' '.join(words)` joins the words with a **space** between them.

Joining with Other Separators

You can use **commas, hyphens, or other characters** instead of spaces.

```
words = ['Python', 'Java', 'C++']
print(', '.join(words))   # Comma separated
print('-'.join(words))    # Hyphen separated
```

OUTPUT

```
Python, Java, C++
Python-Java-C++
```

Handling Punctuation Properly

Sometimes tokenization separates punctuation from words.
We may want to join them **without extra spaces**.

```
tokens = ['Hello', ',', 'world', '!']
# Combine words but avoid space before punctuation
sentence = ''.join([tokens[0]] + [' ' + t if t.isalnum() else t for t in tokens[1:]])
print(sentence)
```

OUTPUT

Hello, world!

From Nested Lists to Strings

Sometimes, text may be in **nested lists**, like paragraphs of tokenized sentences.

```
paragraph = [['Natural', 'Language', 'Processing'], ['is', 'fun']]
# Convert each sentence
sentences = [' '.join(sentence) for sentence in paragraph]
# Join sentences to form paragraph
text = '. '.join(sentences) + '.'
print(text)
```

OUTPUT

```
Natural Language Processing. is fun.
```

Categorizing and Tagging Words

In **Natural Language Processing (NLP)**, **categorizing and tagging words** means **assigning a grammatical or semantic label** to each word in a sentence.
This helps the computer understand the **role** each word plays — whether it's a **noun, verb, adjective**, etc.

This process is called **Part-of-Speech (POS) Tagging**.

Why Is Tagging Important?

Tagging is used in many NLP applications such as:

- **Text classification**
- **Named Entity Recognition (NER)**
- **Machine translation**
- **Speech recognition**
- **Question answering**

It allows the computer to "understand" how words are functioning in a sentence.

Example:

Sentence: *"The cat sat on the mat."*
Tags:

- The → Determiner (DT)
- cat → Noun (NN)
- sat → Verb (VBD)
- on → Preposition (IN)
- the → Determiner (DT)
- mat → Noun (NN)

What Is a Tagger?

- A **Tagger** is an NLP tool or algorithm that automatically assigns tags to words based on their **context and grammar**.
- It takes a sentence as **input** and returns a list of **(word, tag)** pairs as **output**.

Types of Taggers

(a) Rule-Based Taggers

- Use **handwritten grammatical rules** to assign tags.
- Example: If a word ends with "-ed," it is likely a **past tense verb (VBD)**.
- Example tool: **ENGCG (English Constraint Grammar)**

(b) Stochastic Taggers (Statistical Taggers)

- Use **probability and statistics** based on a **trained corpus**.
- Example methods:
    - **Hidden Markov Model (HMM)** Tagger
    - **N-gram Tagger**

These taggers predict the most likely tag based on the previous tags and word frequencies.

(c) Transformation-Based Taggers

- Also known as **Brill Tagger**.
- Starts with simple tagging (e.g., most frequent tag) and **learns transformation rules** to improve accuracy based on errors.

(d) Neural Network Taggers

- Use **Deep Learning models** (e.g., BiLSTM, CRF, Transformers).
- These capture **contextual meaning** of words more accurately.
- Example: **BERT**, **spaCy**, or **NLTK's neural taggers**.

Example Using NLTK (Python)

Here's how tagging works programmatically

```
import nltk
from nltk import word_tokenize
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

text = "The cat sat on the mat"
tokens = word_tokenize(text)
tags = nltk.pos_tag(tokens)

print(tags)
```

OUTPUT

```
[('The', 'DT'), ('cat', 'NN'), ('sat', 'VBD'), ('on', 'IN'), ('the', 'DT'), ('mat', 'NN')]
```

Each pair (word, tag) shows the **category** assigned by the tagger.

Common POS Tags (Penn Treebank Tagset)

Tag	Meaning	Example
NN	Noun (singular)	book
NNS	Noun (plural)	books
VB	Verb (base form)	eat
VBD	Verb (past tense)	ate
JJ	Adjective	beautiful
RB	Adverb	quickly
PRP	Pronoun	he, she
IN	Preposition	on, at
DT	Determiner	the, a

Tagged Corpora (Definition)

A **Tagged Corpus** (plural: *Corpora*) is a **collection of text** where **each word is annotated (tagged)** with its **part of speech (POS)** or other linguistic information.

In simple words:
A tagged corpus = **Text + Tags**

Example

Word	Tag
The	DT
cat	NN
sat	VBD
on	IN
the	DT
mat	NN

Purpose of Tagged Corpora

Tagged corpora are used to:

1. **Train POS Taggers** — taggers learn patterns of how words and tags co-occur.
2. **Evaluate NLP models** — used as a benchmark to check tagging accuracy.
3. **Linguistic analysis** — to study grammar, syntax, and word usage in real language.

Types of Tagged Corpora

1. Part-of-Speech (POS) Tagged Corpora

Each word is tagged with its grammatical category.

Example (NLTK Brown Corpus):

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

2. Morphologically Tagged Corpora

Each word is tagged with **morphological features**, such as:

- Tense (past/present)
- Number (singular/plural)
- Gender (masculine/feminine)

Example:

"sat" → Verb, Past Tense
"cats" → Noun, Plural

3. Syntactically Tagged Corpora (Parsed Corpora)

- Contain **phrase structure** or **dependency structure** information.
- Used for **parsing** and **grammar learning**.

Example (Parse tree):

```
(S
  (NP (DT The) (NN cat))
  (VP (VBD sat)
    (PP (IN on)
      (NP (DT the) (NN mat)))))
```

4. Semantically Tagged Corpora

- Words are tagged with **semantic roles** or **meanings** (like “Agent”, “Action”, “Object”).
- Used in **Semantic Role Labeling (SRL)** and **information extraction**.

Example:

“Ram ate an apple.”
→ Ram (Agent), ate (Action), apple (Object)

Examples of Famous Tagged Corpora

Corpus Name	Description	Language
Brown Corpus	First large-scale tagged corpus (1 million words)	English
Penn Treebank	POS + syntactic annotations, widely used	English
Wall Street Journal (WSJ) Corpus	Subset of Penn Treebank	English
TIMIT	Tagged with phonetic and speech data	English

Corpus Name	Description	Language
Indian Languages Corpora Initiative (ILCI)	Multilingual corpus (Hindi, Tamil, etc.)	Indian Languages
Universal Dependencies (UD)	Cross-linguistic tagged corpus with syntactic & POS info	Multiple

Tagged Corpora in NLTK

NLTK (Natural Language Toolkit) provides many tagged corpora you can use for training or testing taggers.

Example:

```
import nltk
nltk.download('brown')
from nltk.corpus import brown

# Get tagged sentences
tagged_sentences = brown.tagged_sents()
print(tagged_sentences[0])
```

OUTPUT

[('The', 'AT'), ('Fulton', 'NP-TL'), ('County', 'NN-TL'), ('Grand', 'JJ-TL'), ('Jury', 'NN-TL'), ('said', 'VBD'), ...]

How Tagged Corpora Are Used

Step	Purpose
1. Collect text data	Large samples of written/spoken language
2. Annotate words	Linguists or algorithms add tags
3. Train taggers	Machine Learning models learn from these patterns
4. Test accuracy	Compare predicted tags with tagged corpus
5. Apply to real data	Use taggers on untagged sentences

Mapping Words to Properties Using Python Dictionaries

This concept connects **linguistic data (words)** with their **associated features or properties** — and Python **dictionaries** are the perfect structure for this.

---

1. What Does "Mapping Words to Properties" Mean?

In **Natural Language Processing (NLP)**, we often need to store **information about words** — such as:

- Their **Part of Speech (POS)**
- **Lemma** (base form)
- **Meaning or Synonym**
- **Frequency**
- **Word Category** (noun, verb, adjective)
- **Semantic information** (like sentiment, domain, etc.)

To do this efficiently, we **map each word** to its **properties** using a **dictionary**, where:

Key = Word
Value = Property/Properties

Example:

```
word_properties = {
    "run": {"POS": "verb", "tense": "base", "meaning": "move swiftly"},
    "beautiful": {"POS": "adjective", "degree": "positive", "meaning": "pleasing"},
    "dogs": {"POS": "noun", "number": "plural", "base": "dog"}
}

print(word_properties["run"]["POS"])
```

Output:

Verb

Why Use Dictionaries in NLP?

Python dictionaries provide:

- **Fast lookups** → `O(1)` access time
- **Structured storage** for linguistic attributes
- **Flexibility** → can store multiple features per word

Real-World Uses of Word-to-Property Mapping

Application	Description
POS Tagging	Store which tag each word gets (NN, VB, etc.)
Lemmatization	Map inflected forms → base form (e.g., "ran" → "run")
Word Sense Disambiguation	Store different meanings (e.g., "bank" = river side or financial institution)
Sentiment Analysis	Map words to polarity (positive/negative)
Named Entity Recognition (NER)	Map words to entity type (Person, Location, Organization)

Example: Lemmatization Mapping

```
lemmatization_dict = {
    "running": "run",
    "ate": "eat",
    "children": "child",
    "better": "good"
}

word = "children"
print("Base form:", lemmatization_dict[word])
```

Output

```
Base form: child
```

Automatic Tagging

What Is Automatic Tagging?

In **Natural Language Processing (NLP)**, **Automatic Tagging** means **assigning tags (like parts of speech, named entities, etc.) to words automatically using algorithms or trained models** — without manual human labeling.

It's the process of letting the **computer decide the grammatical or semantic role** of each word based on **rules**, **statistics**, or **machine learning**.

Example

Input Sentence:

"The cat sat on the mat."

Automatic Tagger Output:

```
[('The', 'DT'), ('cat', 'NN'), ('sat', 'VBD'), ('on', 'IN'), ('the', 'DT'), ('mat', 'NN')]
```

Here, the **tagger automatically** labeled each word with its **Part of Speech (POS)** tag.

How Automatic Tagging Works

Automatic tagging systems use different methods depending on complexity:

Step-by-step process:

1. **Input Sentence** → “She is playing football.”
2. **Tokenization** → `["She", "is", "playing", "football", "."]`
3. **Model checks each word**:
   - Looks up word in a dictionary or corpus.
   - Checks surrounding words (context).
   - Predicts the most likely tag.
4. **Output** → `[('She', 'PRP'), ('is', 'VBZ'), ('playing', 'VBG'), ('football', 'NN'), ('.', '.')]`

Approaches to Automatic Tagging

There are **three major approaches** to automatic tagging:

A. Rule-Based Tagging

- Uses **handcrafted grammatical rules** and **lexicons**.
- Example rules:
  - If a word ends with **“-ed”**, tag it as **past tense verb (VBD)**.
  - If a word comes after a **determiner (DT)**, tag it as **noun (NN)**.

Example:

```
If (word.endswith('ed')):
    tag = 'VBD'
elif (previous_tag == 'DT'):
    tag = 'NN'
```

Pros: Accurate for small, grammatically clean datasets.
Cons: Hard to scale; requires expert rules.

B. Statistical Tagging (Probabilistic Tagging)

Uses **statistics and probabilities** learned from a **tagged corpus** (like Brown or Penn Treebank).

- Most common: **Hidden Markov Model (HMM)** or **N-Gram Taggers**.
- Each word is tagged based on the **probability** of a tag given the word and its context.

Formula (simplified):

$$P(tag|word) = \frac{P(word|tag) \times P(tag)}{P(word)}$$

Example:
If in training data:

- "sat" appears as a **verb (VBD)** 95% of the time,
  then the tagger will likely assign "sat → VBD".

Pros: Learns from real data.
Cons: Needs a large tagged corpus.

C. Machine Learning / Neural Network Tagging

Modern NLP uses **deep learning models** like:

- **BiLSTM (Bidirectional LSTM)**
- **CRF (Conditional Random Fields)**
- **Transformer models (BERT, RoBERTa, etc.)**

These models learn **contextual patterns** from millions of examples — so they can understand that:

"book" in "I will book a ticket" → **verb**
"book" in "I read a book" → **noun**

Pros: Very accurate, handles ambiguity
Cons: Needs computational resources and training data.

Example Using NLTK (Python)

```
import nltk
from nltk import word_tokenize
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')

sentence = "The quick brown fox jumps over the lazy dog"
tokens = word_tokenize(sentence)
tags = nltk.pos_tag(tokens)

print(tags)
```

Output

```
[('The', 'DT'), ('quick', 'JJ'), ('brown', 'JJ'),
 ('fox', 'NN'), ('jumps', 'VBZ'), ('over', 'IN'),
 ('the', 'DT'), ('lazy', 'JJ'), ('dog', 'NN')]
```

This is Automatic Tagging **in action — done using NLTK's** pre-trained tagger **(Averaged Perceptron Tagger).**

Automatic Tagger Types in NLTK

Tagger	Description
DefaultTagger	Assigns a single default tag to all words (e.g., NN)
RegexTagger	Uses regular expressions for rule-based tagging
UnigramTagger	Assigns tag based on most common tag of the word (from corpus)
Bigram/TrigramTagger	Considers previous one/two tags for context
BrillTagger	Transformation-based learner (hybrid of rule & statistics)

Advantages of Automatic Tagging

- Saves time (vs manual tagging)
- Scalable to millions of words
- Improves consistency
- Can adapt to new languages withtraining
- Used in most real-world NLP systems

Challenges / Limitations

- Ambiguity— words like *"bank"* (river bank or financial bank)
- Unknown words— words not seen in training data
- Context sensitivity— "light rain" (adjective) vs "light the lamp" (verb)

N-Gram Tagging

What Is N-Gram Tagging?

N-Gram Tagging is a **statistical approach** to **automatic tagging** in NLP.
It assigns **Part-of-Speech (POS) tags** to words based on **the tag(s) of the previous (N−1) word(s)** in a sentence.

In simple terms:

An **N-Gram Tagger** uses **context** — the tags of nearby words — to predict the correct tag for the current word.

It's based on the idea that the tag of a word depends not only on the word itself but also on the tags of surrounding words.

What Is an N-Gram?

An **N-Gram** is a **sequence of N items (words or tags)** that appear together.

N	Example	Called As
1	"cat"	**Unigram**
2	"the cat"	**Bigram**
3	"the black cat"	**Trigram**

In tagging, we use **tag sequences** instead of word sequences:

- Unigram Tagger → Uses only the **current word**
- Bigram Tagger → Uses **previous word's tag**
- Trigram Tagger → Uses **previous two tags**

How N-Gram Tagging Works

Step-by-step process:

Let's take a simple sentence:

"The cat sat on the mat"

1. **Training Phase**
   - The tagger is trained on a **tagged corpus** (e.g., Brown or Penn Treebank).
   - It learns how likely a certain **tag sequence** occurs.
   - For example:

        - `P(NN | DT)` = Probability of a Noun (NN) coming after a Determiner (DT).
        - `P(VBD | NN)` = Probability of a Past Tense Verb after a Noun.
2. **Tagging Phase**
    - For each new word, the model selects the tag with the **highest probability**, given the previous (N−1) tags.

Example of Bigram Tagging:

Word	Possible Tags	Previous Tag	Selected Tag
The	DT	—	DT
cat	NN, VB	DT	NN (since NN follows DT often)
sat	NN, VBD	NN	VBD (verb likely after noun)
on	IN	VBD	IN
the	DT	IN	DT
mat	NN	DT	NN

Final Output:

```
[('The', 'DT'), ('cat', 'NN'), ('sat', 'VBD'), ('on', 'IN'), ('the', 'DT'), ('mat', 'NN')]
```

N-Gram Tagging in NLTK

NLTK provides built-in taggers for unigram, bigram, and trigram tagging.

Example Code:

```
import nltk
from nltk.corpus import brown
nltk.download('brown')
nltk.download('universal_tagset')

# Get tagged sentences from the Brown corpus
train_data = brown.tagged_sents(tagset='universal')[:5000]
test_data = brown.tagged_sents(tagset='universal')[5000:5500]

# Train taggers
unigram_tagger = nltk.UnigramTagger(train_data)
bigram_tagger = nltk.BigramTagger(train_data, backoff=unigram_tagger)

# Test the tagger
print(bigram_tagger.tag(['The', 'cat', 'sat', 'on', 'the', 'mat']))
```

OUTPUT

```
[('The', 'DET'), ('cat', 'NOUN'), ('sat', 'VERB'), ('on', 'ADP'), ('the', 'DET'), ('mat', 'NOUN')]
```

Comparison of N-Gram Taggers

Type	Uses	Pros	Cons
Unigram Tagger	Only current word	Fast, simple	Ignores context
Bigram Tagger	Current + previous tag	Context-aware	Fails with unseen pairs
Trigram Tagger	Current + previous two tags	More context	Needs lots of data

Example Comparison

Sentence: “Time flies like an arrow”

Word	**Unigram**	**Bigram**	**Trigram**
Time	NN	NN	NN
flies	NNS	VBZ	VBZ
like	IN	IN	IN
an	DT	DT	DT
arrow	NN	NN	NN

Here, the **Bigram/Trigram taggers** help correctly identify *“flies”* as a **verb (VBZ)**, not a **noun (NNS)**, because of context.

Applications of N-Gram Tagging

- **Part-of-Speech Tagging**
- **Named Entity Recognition (NER)**
- **Speech Recognition**
- **Spell Correction**
- **Text Prediction and Autocomplete**

Transformation-Based Tagging (TBL) — also known as Brill Tagging

Transformation-Based Tagging is a **rule-based approach** to **Part-of-Speech (POS) tagging** in **Natural Language Processing (NLP)**.
It was introduced by **Eric Brill (1995)** and is one of the most famous hybrid methods because it combines both **statistical** and **rule-based** approaches.

Idea Behind TBL

- Instead of directly assigning the best possible tag using probabilities (like HMMs or n-grams),
  TBL **starts with an initial (baseline) tagging** and **gradually improves it** by learning a sequence of transformation rules.
- These rules **correct errors** in the initial tagging step-by-step.

How Transformation-Based Tagging Works

1. **Initialization (Baseline Tagging)**
   - Start by giving each word its **most likely tag** (for example, using unigram statistics — the most frequent tag for each word in the training corpus).
   - Unknown words may get a default tag like **'NN' (noun)**.

   Example:

```
Input sentence: The cat sat on the mat.
Initial tags:   DT NN VBD IN DT NN
```

2. **Learning Transformation Rules**

- The system compares the **current tags** with the **correct tags** (from a tagged corpus).
- It identifies **errors** and learns **rules** that can correct them.
- Each rule has the form:
  "**Change tag A to tag B when condition C is true.**"

Example Rules:

- Change **NN → VB** if the word is preceded by **'to'**
- Change **VBD → VBN** if the word **ends with '-ed'**

3. **Applying the Rules**

- The learned transformation rules are applied **sequentially** to improve tagging accuracy.

- Each rule is applied only if it **reduces the total number of errors**.

4. **Final Output**

- After applying all rules, the output tags are much more accurate than the initial ones.

Example

Suppose we have:

```
Sentence: He can fish.
```

Initial tagging (unigram tagger might produce):

```swift
He/PRP can/MD fish/NN
```

But “fish” here is **a verb**, not a noun.

TBL might learn a rule:

```arduino
Change NN → VB if the previous word is MD (modal verb)
```

After applying this rule:

```swift
He/PRP can/MD fish/VB
```

Advantages

- Combines **accuracy of statistical models** and **interpretability of rule-based systems**.
- Rules are **human-readable**, making debugging and analysis easier.
- Performs well even with **moderate-sized corpora**.

Disadvantages

- **Training is slow** (many rule evaluations).
- **Sequential dependency** — later rules depend on earlier ones.
- May not perform as well as deep learning models on very large datasets.

In NLTK (Python Example)

```
import nltk
from nltk.tbl import demo as brill_demo

# Run the demo Brill Tagger on a sample corpus
brill_demo.demo()
```

This runs a demonstration showing how transformation rules are learned and applied in NLTK.

How to Determine the Category of a Word (Part-of-Speech Tagging in NLP)

In **Natural Language Processing (NLP)**, determining the **category of a word** means identifying its **part of speech (POS)** — for example, whether a word is a *noun, verb, adjective, adverb*, etc.
This process is known as **POS tagging** or **word categorization**.

What is Word Category?

Each word in a sentence belongs to a **syntactic category** (also called a *grammatical category* or *part of speech*).
Examples include:

- **Noun (NN)** → person, place, thing — *dog, book, India*
- **Verb (VB)** → action or state — *run, eat, is*
- **Adjective (JJ)** → describes a noun — *happy, blue, tall*
- **Adverb (RB)** → modifies verbs or adjectives — *quickly, very*
- **Preposition (IN)** → shows relationship — *in, on, under*
- **Determiner (DT)** → specifies a noun — *the, a, some*
- **Pronoun (PRP)** → replaces a noun — *he, she, it*

Methods to Determine the Category of a Word

There are **four main methods** used in NLP to determine a word's category:

1.Lexical Lookup (Dictionary-Based Tagging)

Each word is looked up in a **lexicon** (dictionary) that lists words and their possible categories.

Example:

Word	Possible Categories
book	NN (noun), VB (verb)
play	VB (verb), NN (noun)

Limitation:

Many words are **ambiguous** — they can belong to multiple categories depending on context (e.g., "book a ticket" vs "read a book").

2.Rule-Based Tagging

This method applies **grammatical rules** and **context** to assign the correct tag.

Example Rules:

- If a word ends with **-ly**, tag it as an **adverb (RB)** → *quickly, slowly*
- If a word comes before a noun, tag it as an **adjective (JJ)** → *beautiful flower*
- If a word comes after a determiner (the, a), tag it as a **noun (NN)** → *the cat*

Example:

```
The/DT old/JJ man/NN walks/VBZ slowly/RB
```

3.Statistical (Probabilistic) Tagging

Uses **probability models** trained on large, manually tagged corpora to predict the most likely tag for each word in context.

Examples:

- **Unigram Tagger:** assigns the most frequent tag for a word.
- **Bigram / Trigram Tagger:** uses the tag(s) of the previous one or two words to predict the current tag.
- **Hidden Markov Model (HMM) Tagger:** uses both emission and transition probabilities.
- **Neural Taggers (e.g., BiLSTM, BERT):** use deep learning to capture complex word and sentence patterns.

Example:

"I saw her duck."

- **Unigram tagger:** may tag *duck* as *NN* (noun)
- **Context-aware tagger:** may tag *duck* as *VB* (verb) depending on context ("her duck to avoid something").

4.Combined (Hybrid) Tagging

Modern NLP systems (like NLTK's `pos_tag()` or spaCy) combine:

- Lexical dictionaries,
- Statistical models,
- And sometimes neural networks
  to achieve high accuracy.

Example in Python (Using NLTK)

```
import nltk
from nltk import pos_tag, word_tokenize

# Example sentence
sentence = "The quick brown fox jumps over the lazy dog"

# Tokenize the sentence
tokens = word_tokenize(sentence)

# POS tagging
tags = pos_tag(tokens)

print(tags)
```

OUTPUT

```
[('The', 'DT'), ('quick', 'JJ'), ('brown', 'JJ'),
 ('fox', 'NN'), ('jumps', 'VBZ'), ('over', 'IN'),
 ('the', 'DT'), ('lazy', 'JJ'), ('dog', 'NN')]
```

Summary Table

Method	Description	Example
Lexical Lookup	Dictionary lookup	*book → NN/VB*
Rule-Based	Uses grammar rules	*word ending with* **-***ly → RB*
Statistical	Uses probabilities from data	*HMM, n-gram models*
Neural / Hybrid	Uses deep learning + context	*BERT, spaCy, etc.*

UNIT – 3

What is Text Classification?

Text classification is the process of assigning predefined **categories or labels** to text documents.
Examples:

- Spam detection → *spam / not spam*
- Sentiment analysis → *positive / negative / neutral*
- News categorization → *sports / politics / tech / business*

Supervised Classification

Concept:

In **supervised learning**, the model is trained using a **labeled dataset**, i.e., data where each text is already tagged with its correct category.

Example training data:

Text	Label
"Great movie, I loved it"	Positive
"Worst film ever"	Negative

The algorithm learns patterns from these examples to classify **new unseen text**.

Steps in Supervised Text Classification:

1. **Data Collection:** Gather text samples and their labels.
2. **Preprocessing:**
    - Tokenization
    - Lowercasing
    - Removing stopwords
    - Stemming/Lemmatization
3. **Feature Extraction:**
   Convert text into numerical form (vectors) using techniques like:
    - **Bag of Words (BoW)**
    - **TF-IDF (Term Frequency–Inverse Document Frequency)**
    - **Word Embeddings (Word2Vec, GloVe)**
4. **Model Training:** Train a classifier (e.g., Naive Bayes, Logistic Regression, SVM).
5. **Prediction:** Classify new, unseen texts.
6. **Evaluation:** Measure accuracy and performance.

Evaluation of Classifiers

To test how well the model performs, we use **evaluation metrics** on test data (data not seen during training).

Confusion Matrix

	Predicted Positive	**Predicted Negative**
Actual Positive	True Positive (TP)	False Negative (FN)
Actual Negative	False Positive (FP)	True Negative (TN)

Metrics:

- **Accuracy** = (TP + TN) / (TP + TN + FP + FN)
  → Overall correctness.
- **Precision** = TP / (TP + FP)
  → Out of predicted positives, how many were correct.
- **Recall** = TP / (TP + FN)
  → Out of actual positives, how many were identified correctly.
- **F1-Score** = 2 × (Precision × Recall) / (Precision + Recall)
  → Harmonic mean of precision and recall.

Naive Bayes Classifiers

Naive Bayes is a **probabilistic classifier** based on **Bayes' Theorem**, assuming that all features (words) are independent of each other (hence "naive").

Bayes' Theorem:

◆ **Bayes' Theorem:**

$$P(C|X) = \frac{P(X|C) \times P(C)}{P(X)}$$

Where:

- C: Class (e.g., Positive, Negative)
- X: Document (text)
- $P(C|X)$: Probability that document X belongs to class C
- $P(X|C)$: Probability of document given the class
- $P(C)$: Prior probability of class
- $P(X)$: Probability of document (same for all classes, ignored in comparison)

Working Example:

Let's classify a new sentence — **"This movie is great."**

We calculate:

$$P(\text{Positive}|\text{sentence}) \quad \text{and} \quad P(\text{Negative}|\text{sentence})$$

Whichever is **higher**, that label is assigned.

Types of Naive Bayes:

1. **Multinomial NB:** Used for word counts (common for text classification).
2. **Bernoulli NB:** For binary features (word present/absent).
3. **Gaussian NB:** For continuous data (not common in NLP).

Example (Python-like Logic):

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

texts = ["I love this movie", "This film is terrible", "Amazing acting", "Bad direction"]
labels = ["positive", "negative", "positive", "negative"]

# Convert text to numeric features
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(texts)

# Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(X, labels, test_size=0.5)

# Train Naive Bayes Classifier
model = MultinomialNB()
model.fit(X_train, y_train)

# Predict
predictions = model.predict(X_test)
print("Accuracy:", accuracy_score(y_test, predictions))
```

Advantages:

- Simple and fast to train.
- Works well with small datasets.
- Performs surprisingly well for text classification.

Limitations:

- Assumes word independence (not true in real language).
- Cannot handle very complex relationships between words.

Deep Learning for NLP – Introduction

What is Deep Learning?

Deep Learning (DL) is a branch of **Machine Learning (ML)** that uses **artificial neural networks (ANNs)** with many hidden layers (hence "deep") to automatically learn **representations (features)** from raw data.

In NLP, Deep Learning helps machines understand and generate **human language** — text, speech, and meaning — by learning from large text datasets.

Why Deep Learning for NLP?

Traditional NLP methods (like Bag-of-Words, TF-IDF, or Naive Bayes) rely on **handcrafted features**, which often:

- Ignore word order and context.
- Struggle with large, complex datasets.

Deep Learning solves these by:
□ Learning features automatically from data.
□ Capturing **semantic meaning** (context, relationships, grammar).
□ Handling complex tasks like translation, summarization, and chatbots.

Neural Networks: The Foundation

Basic Structure:

A **neural network** consists of:

1. **Input Layer** – Takes data (e.g., word vectors).
2. **Hidden Layers** – Process features through weighted connections.
3. **Output Layer** – Gives final prediction (e.g., sentiment = positive/negative).

Each connection has a **weight (w)**, and neurons use an **activation function** to introduce non-linearity.

Activation Functions:

They help the network learn complex relationships.

Function	Formula	Range	Purpose
Sigmoid	$\frac{1}{1+e^{-x}}$	(0, 1)	For probabilities
Tanh	$\tanh(x)$	(-1, 1)	Zero-centered activation
ReLU	$\max(0, x)$	[0, ∞)	Faster training, avoids vanishing gradient

How a Neural Network Learns:

1. **Forward Propagation:** Compute output using weights.
2. **Loss Function:** Compare output with true label (error).
3. **Backward Propagation:** Adjust weights to reduce error (using gradient descent).

This iterative process continues until the model's performance improves.

Deep Learning in NLP Tasks

Deep Learning models can handle various NLP tasks such as:

Task	Example	Model Type
Sentiment Analysis	Positive / Negative review	CNN / RNN
Text Classification	Spam / Not Spam	CNN / RNN
Machine Translation	English → French	Seq2Seq (RNN)
Named Entity Recognition	"John lives in Delhi" → (Person, Location)	Bi-LSTM
Chatbots / Question Answering	Conversational AI	Transformer (GPT, BERT)

Word Representation: Word Embeddings

Before feeding text into neural networks, we must **convert words into numbers**.

Word Embedding:

A dense numerical vector that represents a word's **meaning** and **context**.

Example:
"king", "queen", "man", "woman" → vectors close in space if meanings are related.

Common Techniques:

- **Word2Vec** – Learns vector representations from text.
- **GloVe (Global Vectors)** – Uses co-occurrence statistics.
- **FastText** – Considers subword (character-level) information.

These embeddings are the **input features** for deep learning models.

Advantages of Deep Learning in NLP

□ Automatically learns features (no manual feature engineering).
□ Handles large-scale data efficiently.

- Understands**context and sequence** of words.
- Provides**state-of-the-art accuracy** for NLP tasks.

Limitations

- Requires**large datasets** and **computational power**.
- Harder to interpret (black box nature).
- Training can be slow.
- Needs**GPU/TPU** for high performance.

Simple Example Workflow:

```
# Example: Sentiment Classification using Deep Learning

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense

model = Sequential([
    Embedding(input_dim=5000, output_dim=128, input_length=100),  # Word embeddings
    LSTM(64),   # Recurrent layer for sequence learning
    Dense(1, activation='sigmoid')  # Output layer for binary classification
])

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()
```

Convolutional Neural Networks (CNNs)

Introduction:

A **Convolutional Neural Network (CNN)** is a **deep learning model** originally designed for **image processing**, but it also works very well for **text classification** and **NLP tasks**.

CNNs can automatically extract **important local features** (like key phrases or n-grams) from text without requiring manual feature engineering.

Basic Idea

CNNs use a special operation called **convolution**, which slides small filters (kernels) across input data to detect important patterns.

In text, this means:

- Detecting key word patterns (e.g., “not good”, “very bad”)
- Capturing **local dependencies** between nearby words

CNN Architecture for NLP

Let’s go step-by-step

Step 1 – Input Layer

The input is a **sequence of words**, usually converted into **word embeddings**.
Example sentence:

“The movie was really good”

After embedding (say 5 words × 50-dim vector):
→ A **5 × 50** matrix (rows = words, columns = embedding dimensions)

Step 2 – Convolution Layer

- Apply **filters (kernels)** that slide over the word embeddings.
- Each filter detects a specific pattern of nearby words (like a phrase).

Example:

- A filter size of 2 → detects 2-word patterns (“movie was”, “was really”)
- A filter size of 3 → detects 3-word patterns (“The movie was”)

Each filter produces a **feature map** — a numerical representation of detected patterns.

Step 3 – Activation Function

After convolution, an **activation function** (usually **ReLU**) is applied to add non-linearity.

$$\text{ReLU}(x) = \max(0, x)$$

This allows the model to learn complex relationships.

Step 4 – Pooling Layer

Pooling reduces the feature map's size while keeping the most important information.

- **Max Pooling:** Takes the largest value (most important feature).
- **Average Pooling:** Takes the average of the region.

For NLP, **1D Max Pooling** is most common — it helps capture the strongest feature from each filter.

Step 5 – Fully Connected Layer

The pooled features are flattened into a vector and passed through one or more **fully connected (Dense) layers** for final prediction.

Step 6 – Output Layer

Uses **Softmax** (for multi-class) or **Sigmoid** (for binary classification).
Example:

- Sentiment → Positive / Negative
- News category → Sports / Politics / Tech

Example CNN Architecture for Text Classification

```
Input Sentence → Word Embeddings → Convolution Layer
→ ReLU Activation → Max Pooling → Fully Connected Layer
→ Output (e.g., Positive / Negative)
```

Simple Python Example

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Conv1D, GlobalMaxPooling1D, Dense

model = Sequential([
    Embedding(input_dim=5000, output_dim=100, input_length=100),   # Word Embeddings
    Conv1D(filters=128, kernel_size=5, activation='relu'),         # Convolution layer
    GlobalMaxPooling1D(),                                          # Pooling layer
    Dense(1, activation='sigmoid')                                 # Output (binary classificat
])

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()
```

Advantages of CNN in NLP

- Captures**local patterns** (n-grams) efficiently.
- Fast training (parallel computation possible).
- Works well with**short and fixed-length texts**.
- Needs fewer parameters than RNNs.

Limitations

- Cannot easily capture**long-range dependencies** between distant words.
- Not ideal for**sequential context** understanding (for that, use RNNs or Transformers).

Recurrent Neural Networks (RNNs)

Introduction:

A **Recurrent Neural Network (RNN)** is a **deep learning model** designed to handle **sequential data**, where the **order of input matters** — like text, speech, or time series.

Unlike normal neural networks (which treat each input independently), RNNs have a **memory** that captures information from previous inputs.

That makes RNNs ideal for **Natural Language Processing (NLP)** tasks such as:

- Sentence classification
- Machine translation
- Text generation
- Speech recognition

The Need for RNNs in NLP

Text is **sequential** — the meaning of a word depends on previous words.

Example:

"He went to the bank to deposit money."
"He sat on the bank of the river."

The word *"bank"* has different meanings depending on the **previous words**.
So, we need a model that can remember **past context** — that's what RNNs do.

Basic Working

An RNN processes an input sequence **one element (word) at a time**, while maintaining a **hidden state** that stores information about previous steps.

At each time step t:

$$h_t = f(W_h \cdot h_{t-1} + W_x \cdot x_t + b)$$

$$y_t = W_y \cdot h_t + c$$

Where:

- x_t: input at time t (e.g., word embedding)
- h_t: hidden state (memory) at time t
- y_t: output at time t
- W_x, W_h, W_y: weight matrices
- f: activation function (usually tanh or ReLU)

Recurrent Connection

The key feature:
The hidden state hth_tht depends on **both** current input and **previous state** ht−1h_{t-1}ht−1.
That's why it's called **"recurrent"** — the network loops over time steps.

Unfolded RNN Representation

```
x1 → [RNN Cell] → h1 → y1
x2 → [RNN Cell] → h2 → y2
x3 → [RNN Cell] → h3 → y3
...
```

Each RNN cell passes its hidden state **to the next — maintaining sequential memory.**

Types of RNNs

Type	Description	Example Use
One-to-One	Standard NN	Image classification
One-to-Many	One input → Sequence output	Image captioning
Many-to-One	Sequence input → One output	Sentiment analysis
Many-to-Many	Sequence input → Sequence output	Translation, Speech recognition

Problems with Basic RNNs

Vanishing Gradient Problem:

When training long sequences, gradients (error signals) become very small — the model **forgets long-term dependencies**.

Hence, basic RNNs are not good at remembering context far back in the sequence.

Solutions: LSTM and GRU

To fix memory loss, two advanced RNN variants were introduced:

LSTM (Long Short-Term Memory):

- Uses **gates (input, forget, output)** to control information flow.
- Can remember information for **longer sequences**.

GRU (Gated Recurrent Unit):

- A simplified LSTM with fewer gates (update and reset).
- Faster to train, performs similarly well.

Applications

Task	Example
Sentiment Analysis	Predict positive/negative review
Text Generation	Generate new sentences or poetry
Machine Translation	English → French
Named Entity Recognition (NER)	Detect names, places, etc.
Speech Recognition	Convert audio → text

Example RNN Architecture in Python

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, SimpleRNN, Dense

model = Sequential([
    Embedding(input_dim=5000, output_dim=128, input_length=100),  # Word embeddings
    SimpleRNN(64, activation='tanh'),                             # RNN layer
    Dense(1, activation='sigmoid')                                # Output (binary)
])

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()
```

Advantages of RNNs

- Can handle sequential data and context.
- Useful for variable-length inputs.
- Effective in NLP tasks like translation and speech.

Limitations

- Difficult to train on long sequences (vanishing gradient).
- Slow (can’t be fully parallelized).
- Forget distant context.

(Solved by **LSTM** and **GRU**, and later by **Transformers**)

Classifying Text with Deep Learning

What Is Text Classification?

Text classification is the process of assigning a **label or category** to a given text using machine learning or deep learning techniques.

Examples:

- Spam Detection → *Spam / Not Spam*
- Sentiment Analysis → *Positive / Negative*
- News Categorization → *Sports / Politics / Tech*
- Intent Detection → *Booking / Inquiry / Complaint*

Why Deep Learning for Text Classification?

Traditional ML models (Naive Bayes, SVM, Logistic Regression) rely on **hand-crafted features** such as Bag-of-Words or TF-IDF.
These fail to capture:

- **Context** between words
- **Word order**
- **Long-range dependencies**

Deep Learning models (CNNs, RNNs, LSTMs, Transformers) solve this by **automatically learning hierarchical and contextual features** from text.

Deep Learning Workflow for Text Classification

Let’s go step-by-step

Step 1 – Data Preparation

- Collect labeled dataset (text + label).
- Example:

Text	**Label**
“The movie was excellent”	Positive
“I hated the acting”	Negative

- Clean text (remove punctuation, lowercase, etc.).
- Split into **training** and **test** sets.

Step 2 – Text Representation

Convert text into numerical form using:

- **Word Embeddings** (Word2Vec, GloVe, FastText)
- Or use **Embedding Layer** in deep learning frameworks like TensorFlow/Keras.

Each word becomes a dense vector (e.g., 100 dimensions) capturing its meaning.

Step 3 – Model Selection

Depending on the nature of your data, choose a deep learning model:

Model	Strength	Typical Use
CNN	Captures local n-gram patterns	Short text / phrase classification
RNN / LSTM / GRU	Captures sequential context	Long sentences / time-based data
Hybrid CNN + LSTM	Combines local + sequential features	Sentiment analysis, reviews
Transformers (BERT, GPT)	Captures global attention & context	State-of-the-art NLP tasks

Step 4 – Training the Model

1. Feed word embeddings into the network.
2. Network learns to map patterns → labels.
3. Use **loss function** like Binary Cross-Entropy or Categorical Cross-Entropy.
4. Optimize weights via **backpropagation** using optimizers like Adam or SGD.

Step 5 – Evaluation

After training, evaluate performance on the test set using:

- **Accuracy**
- **Precision**
- **Recall**
- **F1-Score**

Example: CNN-Based Text Classifier

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, Conv1D, GlobalMaxPooling1D, Dense

model = Sequential([
    Embedding(input_dim=5000, output_dim=128, input_length=100),   # Word embeddings
    Conv1D(128, 5, activation='relu'),                              # Convolution layer
    GlobalMaxPooling1D(),                                           # Pooling
    Dense(1, activation='sigmoid')                                  # Output layer
])

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()
```

Used for binary classification (e.g., positive vs. negative).

Example: LSTM-Based Text Classifier

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense

model = Sequential([
    Embedding(input_dim=5000, output_dim=100, input_length=100),
    LSTM(64),
    Dense(1, activation='sigmoid')
])

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

Used for longer text or sequence-dependent tasks.

Advanced Approach: Transformers

Modern models like **BERT**, **RoBERTa**, and **GPT** use **self-attention** to understand relationships between all words in a sentence simultaneously.
They achieve **state-of-the-art accuracy** in most NLP classification tasks.

Example task:

BERT fine-tuned for sentiment analysis or spam detection.

Advantages of Deep Learning for Text Classification

- Learns complex patterns automatically.
- Captures context and sequence of words.
- Performs better on large datasets.
- Can be fine-tuned for domain-specific tasks.

Limitations

- Requires large labeled datasets.
- High computational cost (needs GPU).
- Longer training time.
- Harder to interpret (“black box” models).

UNIT – 4

Information Extraction (IE): Overview

Definition:
Information Extraction (IE) is the process of **automatically identifying structured information** (facts, entities, relationships) from **unstructured text** data such as articles, blogs, reviews, or social media posts.

In simple words —

IE converts **raw text** into **structured data** that computers can understand and use.

Example

Input (Unstructured Text):

"Elon Musk founded SpaceX in 2002 and became the CEO of Tesla in 2008."

Output (Structured Information):

Entity 1	Relation	Entity 2	Date
Elon Musk	founded	SpaceX	2002
Elon Musk	became CEO of	Tesla	2008

Steps in Information Extraction

1. **Text Preprocessing**
    - Tokenization (splitting into words/sentences)
    - Stopword removal
    - Lemmatization or Stemming
2. **Part-of-Speech (POS) Tagging**
    - Identifies the grammatical role of words (noun, verb, adjective, etc.)
3. **Named Entity Recognition (NER)**
    - Finds names of **persons, organizations, locations, dates**, etc.
    - Example: “Apple” → Organization, “Tim Cook” → Person

4. **Chunking / Shallow Parsing**
    - Groups words into **phrases** (like Noun Phrases or Verb Phrases)
    - Example: “the red car” → [NP the red car]
5. **Relation Extraction**
    - Determines **relationships between entities** (e.g., *works for*, *located in*, *founded by*).
6. **Template Filling**
    - Extracted entities and relations are placed into **predefined templates** or **structured formats**.

Applications of Information Extraction

- **Search Engines** – Extract key facts for quick answers.
- **Question Answering Systems** – e.g., Chatbots using structured info.
- **Business Intelligence** – Extract company, product, and price data.
- **Social Media Monitoring** – Identify opinions, trends, or named entities.
- **Medical Text Mining** – Extract disease, drug, and symptom relationships.

Techniques Used

Method	Description
Rule-Based Systems	Use hand-written patterns or regex (e.g., “founded by”)
Statistical Models	Use machine learning with annotated data
Deep Learning Models	Use neural networks (e.g., BiLSTM, BERT) for NER and relation extraction

What is Chunking?

Definition:
Chunking (also called **shallow parsing**) is the process of **grouping words** into **meaningful phrases** (like noun phrases or verb phrases) based on their **Part-of-Speech (POS) tags**.

While POS tagging labels individual words, **chunking combines them into higher-level units**.

Example

Sentence:

"The quick brown fox jumps over the lazy dog."

POS Tags:

The/DT quick/JJ brown/JJ fox/NN jumps/VBZ over/IN the/DT lazy/JJ dog/NN

Noun Phrase (NP) Chunking Output:

[NP The quick brown fox] [VP jumps] [PP over] [NP the lazy dog]

Purpose of Chunking

Chunking helps extract **structured information** by:

- Identifying **phrases** (like subjects, objects, etc.)
- Simplifying **sentence structure** for further tasks
- Preparing text for **Named Entity Recognition (NER)** or **Relation Extraction**

Types of Chunks

Type	Example	Description
NP (Noun Phrase)	*The red car*	A noun with its modifiers
VP (Verb Phrase)	*is running fast*	Verb with adverbs or auxiliaries
PP (Prepositional Phrase)	*in the park*	Preposition with a noun phrase
ADJP (Adjective Phrase)	*very beautiful*	Adjectives with modifiers

Chunking Process

1. **Tokenization** → Break text into words
2. **POS Tagging** → Assign parts of speech
3. **Apply Chunking Rules** → Define patterns using regular expressions based on POS tags
4. **Chunk Extraction** → Identify and group phrases

Example in Python (using NLTK)

```
import nltk
from nltk import word_tokenize, pos_tag, RegexpParser

# Input text
text = "The quick brown fox jumps over the lazy dog"

# Tokenize and tag
tokens = word_tokenize(text)
tagged = pos_tag(tokens)

# Define a chunk grammar (for noun phrases)
grammar = "NP: {<DT>?<JJ>*<NN>}"

# Create a chunk parser
parser = RegexpParser(grammar)

# Parse and visualize
chunked = parser.parse(tagged)
chunked.draw()
```

This will show a tree structure **grouping the words into a** noun phrase (NP)**.**

Evaluating Chunkers

When you train a chunker using annotated data, you can evaluate its performance using:

Metric	Description
Precision	% of correctly predicted chunks out of all predicted chunks
Recall	% of correctly predicted chunks out of all actual chunks
F1 Score	Harmonic mean of precision and recall

Chunking vs Parsing

Aspect	Chunking	Full Parsing
Depth	Shallow (phrases only)	Deep (full grammatical structure)
Speed	Fast	Slower
Purpose	Identify key groups (NP, VP)	Understand full syntax tree

Applications

- **Information Extraction** (e.g., identifying "organization names")
- **Named Entity Recognition (NER)**
- **Question Answering Systems**
- **Machine Translation**
- **Text Summarization**

What is a Chunker?

A **chunker** is a model or a rule-based system that **automatically detects and groups phrases** (like noun phrases, verb phrases) in a sentence after **POS tagging**.

In short:
Chunking = POS tagging + Pattern recognition for phrases

You can **develop** a chunker using:

- **Rule-based (Grammar/Regex)** approach
- **Machine learning-based** approach (trained chunkers)

Developing a Chunker

There are two main ways:

A. Rule-Based Chunker (Using Regular Expressions)

We define patterns using **POS tags** to identify chunks.

Example:

```
import nltk
from nltk import word_tokenize, pos_tag, RegexpParser

# Sample text
sentence = "The beautiful red car was parked near the university"

# Tokenize and tag
tokens = word_tokenize(sentence)
tagged = pos_tag(tokens)

# Define grammar for Noun Phrase (NP)
grammar = "NP: {<DT>?<JJ>*<NN>}"

# Create a chunk parser
chunk_parser = RegexpParser(grammar)

# Apply the chunker
chunked = chunk_parser.parse(tagged)

# Display chunks
print(chunked)
chunked.draw()
```

Explanation:

- `<DT>?` → Optional Determiner (like *the*, *a*, *an*)
- `<JJ>*` → Zero or more adjectives
- `<NN>` → Noun
  So this rule captures noun phrases like *"The beautiful red car"*.

B. Machine Learning-Based Chunker

Uses **supervised learning** — you train a model with:

- **Input:** POS-tagged sentences
- **Output:** Chunk labels (e.g., "B-NP", "I-NP", "O")

Example using NLTK's built-in dataset:

```
from nltk.corpus import conll2000
from nltk.chunk import ChunkParserI
from nltk.chunk.util import tree2conlltags
from nltk.tag import UnigramTagger

# Load data
train_sents = conll2000.chunked_sents('train.txt', chunk_types=['NP'])
test_sents = conll2000.chunked_sents('test.txt', chunk_types=['NP'])

# Create training data
train_data = [[(t, c) for w, t, c in tree2conlltags(sent)] for sent in train_sents]

# Train a Unigram chunker
class UnigramChunker(ChunkParserI):
    def __init__(self, train_sents):
        train_data = [[(t, c) for w, t, c in tree2conlltags(sent)] for sent in train_sents]
        self.tagger = UnigramTagger(train_data)
    def parse(self, sentence):
        pos_tags = [pos for (word, pos) in sentence]
        tagged_pos = self.tagger.tag(pos_tags)
        chunks = [(word, pos, chunk) for ((word, pos), (pos2, chunk)) in zip(sentence, tagged_pos)]
        return nltk.chunk.conlltags2tree(chunks)
```

This type of model learns patterns automatically **from annotated corpora like** CONLL 2000**.**

Evaluating Chunkers

Once a chunker is developed, its performance must be **evaluated** on a test set.

Evaluation Metrics:

Evaluation Metrics:

Metric	Formula	Meaning
Precision	$P = \frac{TP}{TP+FP}$	% of predicted chunks that are correct
Recall	$R = \frac{TP}{TP+FN}$	% of true chunks that were correctly found
F1-Score	$F1 = \frac{2PR}{P+R}$	Balance between precision and recall

Where:

- **TP (True Positive):** Correctly identified chunks
- **FP (False Positive):** Incorrectly predicted chunks
- **FN (False Negative):** Missed correct chunks

Evaluating in NLTK

```
from nltk.corpus import conll2000
from nltk.chunk import UnigramChunker

train_sents = conll2000.chunked_sents('train.txt', chunk_types=['NP'])
test_sents = conll2000.chunked_sents('test.txt', chunk_types=['NP'])

chunker = UnigramChunker(train_sents)
print(chunker.evaluate(test_sents))
```

OUTPUT

```
ChunkParse score:
    Precision: 86.7%
    Recall: 88.4%
    F-Measure: 87.5%
```

Importance of Evaluation

- Helps measure **accuracy and reliability** of the chunker.
- Allows comparison between **different approaches** (rule-based vs ML).
- Ensures **robustness** for downstream tasks like NER or relation extraction

Recursion in Linguistic Structure

Definition:
In linguistics, **recursion** means a **phrase can contain another phrase of the same type** — this allows language to express **infinite ideas with finite rules**.

In simple words:

Recursion lets sentences **embed smaller sentences or phrases** inside themselves.

Example

1. **Basic sentence:**

"The cat sat."

2. **Add a phrase (recursion in noun phrase):**

"The cat on the mat sat."

3. **Add another phrase:**

"The cat on the mat near the door sat."

Here, each noun phrase ("cat", "cat on the mat", "cat on the mat near the door") **contains another noun phrase** → recursion in structure.

Why Recursion Happens

Language has **hierarchical structure** — a sentence (S) is made up of phrases (NP, VP, PP), and those phrases can contain **other phrases of the same kind**.

For example:

```
S → NP VP
NP → DT N | NP PP
PP → P NP
```

Because NP → NP PP, it allows recursion —
A noun phrase (NP) **can contain a** prepositional phrase (PP)**, and that PP again can contain another NP.**

Example Tree

For the sentence:

"The book on the table in the room is mine."

```
(S
   (NP
      (NP (DT The) (NN book))
      (PP (IN on)
          (NP
             (NP (DT the) (NN table))
             (PP (IN in)
                 (NP (DT the) (NN room))))))
   (VP (VBZ is) (PRP$ mine)))
```

Here you can see:

- NP contains a PP
- That PP contains another NP
- That NP again contains another PP
  → **recursive pattern!**

Importance of Recursion in NLP

Task	**Role of Recursion**
Parsing	Helps build hierarchical syntactic trees.
Information Extraction	Allows extraction from nested phrases.
Machine Translation	Handles nested and dependent clauses correctly.
Question Answering	Helps understand embedded questions.
Text Summarization	Recognizes main vs subordinate clauses.

Recursion in Grammar Rules (CFG)

In **Context-Free Grammars (CFGs)** — used in NLP parsers — recursion appears naturally in rules:

Example:

```
NP → NP PP
PP → P NP
```

If the grammar allows a non-terminal (like NP) to appear on both sides of a rule**, it's** recursive**.**

Recursion in Programming (Python + NLTK Example)

You can visualize recursive linguistic structure using **NLTK's parser**:

```
import nltk
from nltk import CFG

# Define a simple grammar with recursion
grammar = CFG.fromstring("""
  S -> NP VP
  NP -> DT NN | NP PP
  PP -> IN NP
  VP -> VB NP
  DT -> 'the'
  NN -> 'cat' | 'mat' | 'room'
  IN -> 'on' | 'in'
  VB -> 'saw'
""")

parser = nltk.ChartParser(grammar)
sentence = ['the', 'cat', 'on', 'the', 'mat', 'in', 'the', 'room', 'saw', 'the', 'cat']

for tree in parser.parse(sentence):
    print(tree)
    tree.draw()
```

This creates a recursive parse tree — **showing nested NP and PP structures.**

What is Named Entity Recognition (NER)?

Definition:
Named Entity Recognition (NER) is the process of **identifying and classifying named entities** in a text into predefined categories such as **person names, organizations, locations, dates, monetary values**, etc.

In simple words —
NER finds **real-world objects** in text and labels them with their **type**.

Example:

Sentence:

"Elon Musk founded SpaceX in 2002 and lives in Texas."

NER Output:

Entity	Type
Elon Musk	PERSON
SpaceX	ORGANIZATION
2002	DATE
Texas	LOCATION

Steps in Named Entity Recognition

1. **Text Preprocessing**
   - Tokenization
   - Stopword Removal
   - Lemmatization
2. **Part-of-Speech (POS) Tagging**
   - Identifies grammatical roles (noun, verb, etc.)
3. **NER Tagging**
   - Detects entities and assigns category labels
   - e.g., *New York* → LOCATION, *Google* → ORGANIZATION
4. **Post-Processing**
   - Merge or refine overlapping entities.

Common Named Entity Types

Category	Examples
PERSON	Elon Musk, Narendra Modi
ORGANIZATION	Google, Gurugram University
LOCATION	Delhi, India, Ganga River
DATE/TIME	12th February 2005, 5 PM
MONEY	₹5000, $10 million

Category	Examples
PERCENT	25%, 80 percent
PRODUCT	iPhone, Tesla Model S
EVENT	Olympic Games, World War II

Approaches to NER

A. Rule-Based (Pattern Matching)

- Uses **regular expressions** and **hand-written linguistic rules**.
- Example: Words ending with *Ltd.* → ORGANIZATION
- Works well for simple domains but fails on complex language.

B. Machine Learning-Based

- Train models using **labeled corpora** (supervised learning).
- Uses features like capitalization, word shape, POS tags, etc.
- Common algorithms:
  - Hidden Markov Model (HMM)
  - Conditional Random Fields (CRF)
  - Support Vector Machines (SVM)

C. Deep Learning-Based (Modern NER)

- Uses **neural networks** to automatically learn features from text.
- Common architectures:
  - **BiLSTM + CRF**
  - **CNN + LSTM**
  - **Transformers (BERT, RoBERTa, GPT, etc.)**
- Highly accurate and widely used today.

Example in Python (using spaCy)

```
import spacy

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

# Input text
text = "Elon Musk founded SpaceX in 2002 and lives in Texas."

# Process text
doc = nlp(text)

# Extract entities
for ent in doc.ents:
    print(ent.text, "→", ent.label_)
```

Output

```
Elon Musk → PERSON
SpaceX → ORG
2002 → DATE
Texas → GPE
```

(GPE = Geopolitical Entity, i.e., country, city, or state)

Applications of NER

Application	Example
Information Extraction	Extract company names, dates, and locations from news articles
Question Answering	Identify key entities in user queries
Summarization	Highlight people, places, and organizations in summaries
Search Engines	Improve relevance by recognizing entity names
Chatbots	Understand entities like names, dates, and locations from user messages

What is Relation Extraction (RE)?

Definition:
Relation Extraction (RE) is the process of **detecting and classifying semantic relationships** between **entities** identified in a text.

In simple words —
After NER finds *who* and *what*,
Relation Extraction finds *how they are related.*

Example

Sentence:

"Elon Musk founded SpaceX in 2002."

From NER:

- Elon Musk → PERSON
- SpaceX → ORGANIZATION
- 2002 → DATE

Relation Extraction Output:

Entity 1	Relation	Entity 2	Extra Info
Elon Musk	founded	SpaceX	2002

So, RE helps us capture **(Subject, Relation, Object)** triplets —
→ (Elon Musk, founded, SpaceX)

Steps in Relation Extraction

1. **Preprocessing**
   - Tokenization, POS tagging, and dependency parsing.
2. **Named Entity Recognition (NER)**
   - Identify entities like PERSON, ORGANIZATION, LOCATION, etc.
3. **Relation Detection**
   - Identify **whether a relationship exists** between two entities.
4. **Relation Classification**
   - Classify the **type of relation** (e.g., *founded by*, *born in*, *located in*, etc.).

Types of Relations

Category	Example	Relation Type
Organizational	"Elon Musk founded SpaceX."	founderOf
Geographical	"Taj Mahal is located in Agra."	locatedIn
Personal	"Barack Obama is married to Michelle Obama."	spouseOf
Professional	"Sundar Pichai is CEO of Google."	worksFor
Temporal	"World War II ended in 1945."	endedIn

Approaches to Relation Extraction

A. Rule-Based (Pattern Matching)

- Uses manually defined **patterns** or **regular expressions**.
- Example rule:
  If pattern matches “X founded Y” → Relation = *founderOf*

Example:

“Steve Jobs founded Apple.”
→ (Steve Jobs, founderOf, Apple)

□ Simple but □ fails for complex sentence structures.

B. Supervised Machine Learning

- Uses **annotated datasets** (text with labeled relations).
- Each entity pair becomes a training example.
- **Features used:** POS tags, dependency paths, word distance, etc.
- Common algorithms:
  - Support Vector Machines (SVM)
  - Decision Trees
  - Naive Bayes
  - Logistic Regression

□ More flexible than rules, but □ needs large labeled data.

C. Deep Learning / Neural Models

- Automatically learn features from raw text.
- Common architectures:
  - **CNN** (captures local word patterns)
  - **RNN / LSTM** (captures long dependencies)
  - **Transformer-based models** like **BERT**, **RoBERTa**

Example:
Sentence: “Bill Gates founded Microsoft.”
→ Model output: (Bill Gates, founder_of, Microsoft)

- Very accurate
- Requires high computation and large data.

Relation Extraction Example (using spaCy)

```
import spacy
from spacy.matcher import Matcher

# Load NLP model
nlp = spacy.load("en_core_web_sm")

# Input text
text = "Elon Musk founded SpaceX in 2002."

# Process text
doc = nlp(text)

# Print named entities
for ent in doc.ents:
    print(ent.text, ent.label_)

# Simple pattern-based relation detection
matcher = Matcher(nlp.vocab)
pattern = [{'ENT_TYPE': 'PERSON'}, {'LEMMA': 'found'}, {'ENT_TYPE': 'ORG'}]
matcher.add("FOUNDER_RELATION", [pattern])
```

```
matches = matcher(doc)
for match_id, start, end in matches:
    span = doc[start:end]
    print("Relation:", span.text)
```

OUTPUT

```
Elon Musk PERSON
SpaceX ORG
Relation: Elon Musk founded SpaceX
```

Applications of Relation Extraction

Field	Example
Knowledge Graphs	Build (Entity, Relation, Entity) triples for Google Knowledge Graph
Question Answering Systems	"Who founded Tesla?" → extract (Elon Musk, founderOf, Tesla)
Information Retrieval	Enhance search by linking related entities
Biomedical NLP	Extract relations like (Drug, treats, Disease)
News Analysis	Identify relations between people, events, and organizations

Analyzing Sentence Structure

Analyzing sentence structure in NLP means understanding **how words are organized and related** in a sentence.
It's about **syntax** — the rules and patterns governing how words combine to form **meaningful sentences**.

Before extracting meaning, we need to know **what role each word plays** (subject, verb, object, modifier, etc.) and how phrases are structured.

Some Grammatical Dilemmas

In natural language, many **sentences can be ambiguous** or have structures that are **difficult for computers to parse**. These are called **grammatical dilemmas**.

A. Syntactic Ambiguity

- A sentence can have **more than one valid parse**.
- **Example:**

> "I saw the man with a telescope."

Two interpretations:

1. I used a telescope to see the man.
2. The man I saw had a telescope.

- Computers must **decide which structure is intended**, which is tricky without context.

B. Part-of-Speech Ambiguity

- A word can have **multiple possible POS tags** depending on context.
- **Example:**

> "Book the flight." → *Book* = verb
> "The book is on the table." → *Book* = noun

- NLP systems must **disambiguate words** based on sentence structure.

C. Attachment Ambiguity

- Ambiguity about **which part of the sentence a phrase modifies**.
- **Example:**

> "She saw the boy with the binoculars."

  - Did she have the binoculars?
  - Or did the boy have them?

- This is common with **prepositional phrases (PPs)**.

D. Coordination Ambiguity

- Ambiguity in sentences with **"and," "or," or other conjunctions**.
- **Example:**

  "He saw the man and the woman with a telescope."

  - Does *with a telescope* modify both *man and woman* or just *woman*?

E. Modifier Scope Ambiguity

- Ambiguity arises from **adjectives or adverbs**.
- **Example:**

  "Old men and women were present."

  - Are both men and women old? Or only the men?

F. Ellipsis / Missing Elements

- Some sentences omit words but are still understandable to humans.
- **Example:**

  "I ordered pizza, and John [ordered] pasta."

  - NLP must **infer the missing verb**.

Why These Dilemmas Matter in NLP

- Ambiguities cause **parsing errors**, which affect downstream tasks:
  - **Information Extraction** → Misidentified entities or relations
  - **Machine Translation** → Incorrect translations
  - **Question Answering** → Wrong answers due to misinterpreted structure

- Handling these dilemmas often requires:
  - **Contextual information** (e.g., surrounding sentences)
  - **Probabilistic models** (like probabilistic CFGs)
  - **Deep learning approaches** that learn likely structures

Syntax in NLP

Syntax is the set of rules that governs **how words are combined to form grammatically correct sentences**.

In NLP, syntax helps **analyze the structure of a sentence**, rather than just its words, allowing systems to understand **relationships between words**.

Syntax = the *structure* of the sentence. Semantics = the *meaning* of the sentence.

Why Syntax is Important

Syntax is crucial in NLP because many tasks **cannot rely solely on individual words**. Understanding sentence structure helps in:

1. **Disambiguating Meaning**
   - Example (Attachment ambiguity):

     > "I saw the man with a telescope."
     > Syntax helps determine whether *with a telescope* refers to "I" or "the man".

2. **Information Extraction**
   - Helps extract structured knowledge like entities and relationships.
   - Example:

     > "Elon Musk founded SpaceX."
     > Knowing subject-verb-object structure → (Elon Musk, founded, SpaceX)

3. **Machine Translation**
   - Accurate translation requires understanding **sentence structure**, not just word-by-word translation.
4. **Question Answering & Chatbots**

    - Understanding syntax helps identify **who did what to whom**.
    - Example: “Who founded SpaceX?”
        - Needs subject-verb-object parsing.
5. **Summarization**
    - Syntax helps identify **main clauses** versus subordinate clauses to summarize key information.
6. **Grammar Checking**
    - Detect errors in writing using syntactic rules.

Syntax vs Semantics

Aspect	**Syntax**	**Semantics**
Focus	Structure of sentence	Meaning of sentence
Example	“The cat sat on the mat.”	Understanding that a cat is sitting on a mat
Role in NLP	Parsing, POS tagging, chunking	NER, Relation Extraction, QA

How Syntax is Represented in NLP

1. **Parse Trees**
    - Trees represent **hierarchical structure** of sentences.
    - Example: Noun Phrases (NP), Verb Phrases (VP), Prepositional Phrases (PP).
2. **Context-Free Grammar (CFG)**
    - Defines **rules for generating valid sentences** (we’ll study this in next topic).
3. **Dependency Parsing**
    - Represents **syntactic relationships** as dependencies between words.
    - Example: In “Elon Musk founded SpaceX”, *founded* → root, *Elon Musk* → subject, *SpaceX* → object.

Context-Free Grammar (CFG)

Definition:
A **Context-Free Grammar (CFG)** is a set of rules used to **generate all possible sentences in a language**.
It defines **how words and phrases combine hierarchically** to form valid sentences.

CFG is called “context-free” because **the rules apply regardless of surrounding words**.

Components of a CFG

A CFG consists of **four parts**:

1. **Terminals (Σ)**
   - The actual words in the language.
   - Example: “dog”, “barks”, “the”, “runs”
2. **Non-terminals (N)**
   - Syntactic categories or placeholders for phrases.
   - Example: S (sentence), NP (noun phrase), VP (verb phrase), PP (prepositional phrase)
3. **Start Symbol (S)**
   - Represents a **complete sentence**. Parsing starts from this.
   - Usually `S` is used.
4. **Production Rules (P)**
   - Define **how non-terminals can be expanded** into other non-terminals or terminals.
   - Example:

   ```
   S → NP VP
   NP → DT NN
   VP → VB NP
   ```

How CFG Works (Example)

Goal: Generate the sentence → "The cat sleeps"

Grammar Rules:

```
S  → NP VP
NP → DT NN
VP → VB
DT → 'The'
NN → 'cat'
VB → 'sleeps'
```

Derivation:

```
S
→ NP VP
→ DT NN VP
→ 'The' NN VP
→ 'The' 'cat' VP
→ 'The' 'cat' VB
→ 'The' 'cat' 'sleeps'
```

This shows how a CFG generates a valid sentence step by step**.**

Why CFG is Useful in NLP

1. **Parsing Sentences**
   - Helps build **parse trees** that represent the hierarchical structure of sentences.
2. **Syntax Analysis**
   - Ensures sentences follow grammatical rules.

    - Detects errors or ambiguity.
3. **Supports Downstream NLP Tasks**
    - **Information Extraction** – identify subjects, objects, relations
    - **Machine Translation** – map structure to target language
    - **Question Answering** – understand syntactic relations
4. **Recursive Structures**
    - CFG naturally handles **recursion**, e.g., nested noun phrases or prepositional phrases.

Example CFG Parse Tree

Sentence: **"The cat sat on the mat"**

```
          S
        /   \
      NP     VP
     / \    /  \
   DT  NN  VB   PP
   |   |   |   / \
  The  cat sat  P  NP
                |  / \
               on DT NN
                  |   |
                 the  mat
```

Shows **sentence structure** with NP, VP, PP, DT, NN, VB.

Key Notes

- **CFG is simpler than full natural language grammar** but powerful enough for many NLP tasks.

- Ambiguities still exist — multiple parse trees may be possible.
- Can be **extended with probabilities** → Probabilistic CFG (PCFG), which helps **choose the most likely parse**.

What is Parsing?

Definition:
Parsing is the process of **analyzing the syntactic structure of a sentence** according to a grammar (like CFG).

In NLP, parsing helps determine **how words in a sentence are related** and constructs a **parse tree** showing hierarchical structure.

Why Parsing is Important

1. **Understanding Sentence Structure**
   - Identifies subjects, verbs, objects, and modifiers.
2. **Disambiguation**
   - Resolves structural ambiguity in sentences.
   - Example: "I saw the man with a telescope" → different parse trees for different interpretations.
3. **Supports Downstream NLP Tasks**
   - **Information Extraction** → identifies entities and relationships
   - **Machine Translation** → maps structures between languages
   - **Question Answering** → identifies what action involves which entity

---

How Parsing Works with CFG

Step 1: Start with the Start Symbol

- Typically `S` (sentence)

Step 2: Apply Production Rules

- Expand non-terminals (like NP, VP, PP) using CFG rules

Step 3: Match Terminals

- Continue expansions until all words in the sentence are matched

Step 4: Build Parse Tree

- Each expansion forms a node in the tree
- Leaf nodes are the actual words (terminals)

Example CFG

Grammar:

```
S  → NP VP
NP → DT NN | DT JJ NN
VP → VB NP | VB PP
PP → P NP
DT → 'the'
JJ → 'quick'
NN → 'fox' | 'dog'
VB → 'jumps'
P → 'over'
```

Sentence:

“The quick fox jumps over the dog”

Parse Tree:

```
         S
        / \
      NP     VP
    / | \   / \
  DT   JJ   NN VB   PP
  |    |    |  |   / \
the quick fox jumps P   NP
                    |   / \
                  over DT NN
                       |   |
                      the  dog
```

Types of Parsers

1. **Top-Down Parsing**
   - Start from **start symbol** and try to generate the sentence.
   - Checks if CFG rules can produce the sentence.
2. **Bottom-Up Parsing**
   - Start from **words in the sentence** and try to combine them to form higher-level phrases until reaching the start symbol.
3. **Chart Parsing**
   - Efficient method storing partial parses in a **chart** to avoid redundant computations.
4. **Probabilistic Parsing (PCFG)**
   - Assigns **probabilities to CFG rules**
   - Chooses the **most likely parse tree** in case of ambiguity

Parsing in NLP Tools (Example with NLTK)

```
import nltk
from nltk import CFG

# Define CFG
grammar = CFG.fromstring("""
  S -> NP VP
  NP -> DT JJ NN | DT NN
  VP -> VB NP | VB PP
  PP -> P NP
  DT -> 'the'
  JJ -> 'quick'
  NN -> 'fox' | 'dog'
  VB -> 'jumps'
  P -> 'over'
""")
```

```
# Create parser
parser = nltk.ChartParser(grammar)

sentence = ['the', 'quick', 'fox', 'jumps', 'over', 'the', 'dog']

# Parse and visualize
for tree in parser.parse(sentence):
    print(tree)
    tree.draw()
```

This generates the **parse tree**, showing how the sentence is constructed from the CFG.