UNIT 1

CONTENTS

- Origins and challenges of NLP –
- Language Modelling: Grammar-based LM, Statistical LM –
- Regular Expressions, Finite-State Automata – English Morphology,
- Transducers for lexicon and rules,
- Tokenization, Detecting and Correcting Spelling Errors,
- Minimum Edit Distance

NATURAL LANGUAGE

- A natural language or ordinary language is defined as any language **that occurs naturally in a human community through a process of use, repetition**, and change without conscious planning.

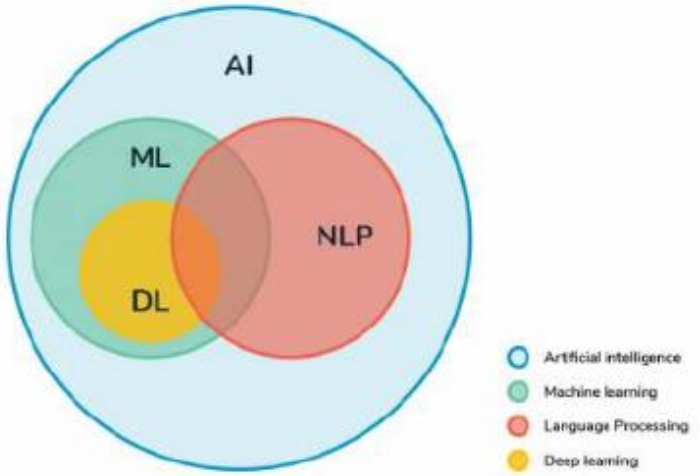
NATURAL LANGUAGE PROCESSING (NLP)

- Natural Language Processing (NLP) refers to the **branch of computer science—and more specifically, the branch of artificial intelligence or AI.**
- It is concerned with **giving computers the ability to understand text and spoken words** in much the same way **human beings can.**
- NLP seeks to enable machines to **interact with humans using natural language**.

NATURAL LANGUAGE PROCESSING (NLP)

Natural language processing (NLP) is

- A field of computer science, artificial intelligence (also called machine learning), and linguistics
- Concerned with the interactions between computers and human (natural) languages.
- Specifically, the process of a computer extracting meaningful information from natural language input and/or producing natural language output ”



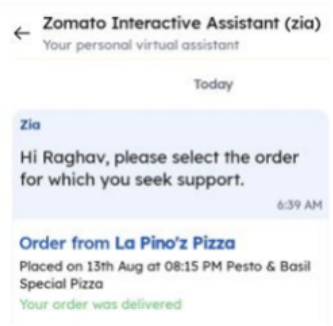
How many of you believe you've interacted with an application powered by Natural Language Processing (NLP) today or in the past week?

Take a moment to think 🤔 about it.

Applications of NLP

1. Zomato Interactive Assistant:

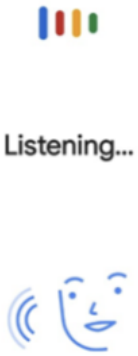
Have you ever ordered food online and chatted with a customer support bot?



This is Zomato's chatbot. It leverages NLP to understand your queries and respond in a human-like manner. These bots can assist with order status, address concerns, or even recommend dishes!

2. Voice Search & Assistants:

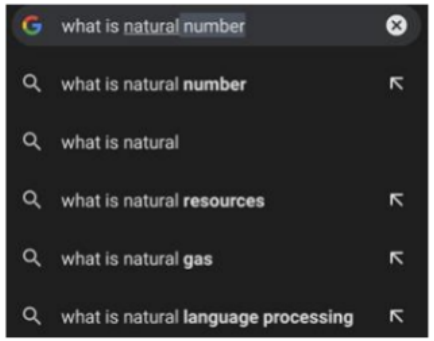
When you say, "Hey Google" or "Siri", and ask a question or give a command, you're using NLP.



Voice-activated assistants rely heavily on NLP to understand and process our spoken language. It's not just about recognizing words, but understanding intent.

3. Autocomplete & Predictive Text:

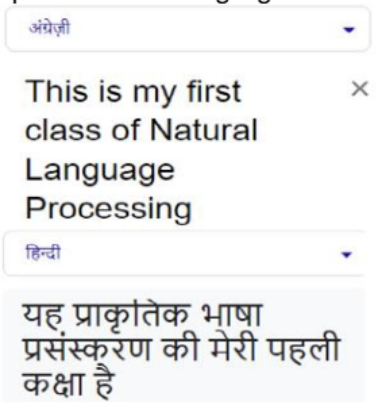
Have you ever started typing a message or search, and your device suggests the next word?



This is NLP in action! Predictive text models anticipate what you're likely to type next based on your history and common language patterns.

4. Translation Tools:

Have you used tools like Google Translate to decipher a phrase in a different language or to communicate with someone who speaks another language?



NLP powers these tools, enabling real-time translations and helping bridge language barriers.

By now, you've probably realized that **you're using NLP more often than you thought! It's embedded in many of our daily tech interactions**. As we proceed in this course,

we'll uncover the magic behind these applications and explore how we can harness the power of NLP for innovative solutions.

Common Applications of Natural Language Processing:

1. Conversational Agents:

You might know these as chatbots. They're programmed to simulate human-like conversation and can be found on many websites offering customer support or assistance.

2. Text Generation:

When you start typing a message and your phone suggests the next word or completes it for you, that's text generation in action. Examples include autocomplete features on search engines and keyboards.

3. Machine Translation:

Tools like Google Translate that can convert text from one language to another almost instantly.

4. Spelling, Grammar, and Writing Correction:

Software like Grammarly checks not just for spelling errors but also for grammatical mistakes and even style and tone in some cases.

5. Text Summarization:

Have you ever wanted a quick summary of a long article? Platforms like Inshorts provide brief 60-word summaries of news articles. Another platform, Artifact, offers AI-driven news summaries.

6. Document Clustering:

Ever wondered how search engines or databases group similar documents together? This process of categorizing documents into "clusters" based on their content or themes is called document clustering.

7. Document Classification:

This is the process of automatically assigning predefined categories or labels to a given text. For instance, an email service might classify emails as "spam" or "not spam."

8. ChatGPT:

A more advanced version of chatbots. These are powered by potent language models like the one you're interacting with right now. They can generate text and answer questions based on vast amounts of data they've been trained on.

And Many More...

NLP is everywhere! From voice assistants on your phone to the algorithms that filter out inappropriate comments on social platforms.

History

Origins of NLP:

NLP (Natural Language Processing) has its roots in several fields, including linguistics, computer science, and artificial intelligence.

1. 1950s – Early Beginnings:

- Alan Turing proposed the Turing Test to evaluate if machines could mimic human intelligence.
- Rule-based systems and symbolic methods were the primary approaches to process language.
- Example: Translating languages (like Russian to English) using grammar rules.

2. 1970s – Grammar and Syntax:

- Focus shifted to understanding the structure of language (syntax and grammar).
- Techniques like Chomsky's formal grammar were applied to model language rules.

3. 1980s – Statistical Models:

- Emergence of probabilistic and statistical approaches due to the availability of more data.
- Hidden Markov Models (HMMs) and language modeling became popular.

4. 1990s – Machine Learning:

- Machine learning techniques, including decision trees and Support Vector Machines (SVMs), were applied to NLP tasks.
- Tasks like speech recognition, part-of-speech tagging, and named entity recognition improved.

5. 2000s – Data-Driven NLP:

- The rise of the internet provided massive datasets for training models.
- Statistical methods dominated, with techniques like n-grams and Latent Semantic Analysis (LSA).

6. 2010s – Deep Learning Revolution:
Neural networks and deep learning transformed NLP, allowing models to understand context better.
Techniques like word embeddings (Word2Vec, GloVe) and transformers (e.g., BERT, GPT) emerged.

7. Present – Large Language Models:
NLP now uses powerful pre-trained models like GPT-4 and BERT to achieve near-human performance in many tasks.

Early Foundations (Pre-1950s — Linguistic Theories)

Before computers could process language, humans were trying to *understand* how language works, using **formal linguistic theories**.

Syntax (Structure):

« Sentence: [Subject] [Verb] [Preposition] [Article] [Object] »

Semantics (Meaning):

Ex: The Suresh and Ramesh are Best friends. => meaning that two persons and their relationshiops.

Noam Chomsky's Generative Grammar (1957)

He introduced **phrase structure rules:**

- S → NP + VP
- NP → Det + N
- VP → V + NP

Example : The tom chases the jerry.

NP = The tom ; VP = chases the jerry ; NP = the jerry ;

Det (determiner) = the

1950s – Early Beginnings

Alan Turing (in his famous 1950 paper
"**Computing Machinery and Intelligence**")
asked:
"Can machines think?"
Turing Test : To **measure machine intelligence through language**.

Computers were just being developed, and scientists began exploring if machines could **understand and process human language**.

1950s – Early Beginnings

Georgetown University (USA) –

IBM Machine Translation Experiment (1954)

- They translated **Russian to English**
- Used **about 250 vocabulary words**

If word == "дом", then word = "house“

"Машина работает хорошо"

("machine works well")

Grammar NLP

This was the time when NLP started focusing on the **formal structure of language** using **grammatical rules**.

“A good sentence in English looks like:
Subject + Verb + Object
Like: ‘Ravi eats mango.’”

Just like you, computers in the 1970s were **taught grammar** using **formal rules** — just like teaching a child.

Grammar NLP

Grammar as Rules (Chomsky's formal grammar)

Sentence → Noun Phrase + Verb Phrase
Noun Phrase → Determiner + Noun
Verb Phrase → Verb + NounPhrase
These are called **Context-Free Grammar (CFG)** rules.

Grammar NLP

Lexicon (Word Dictionary)

We create a list of known words:

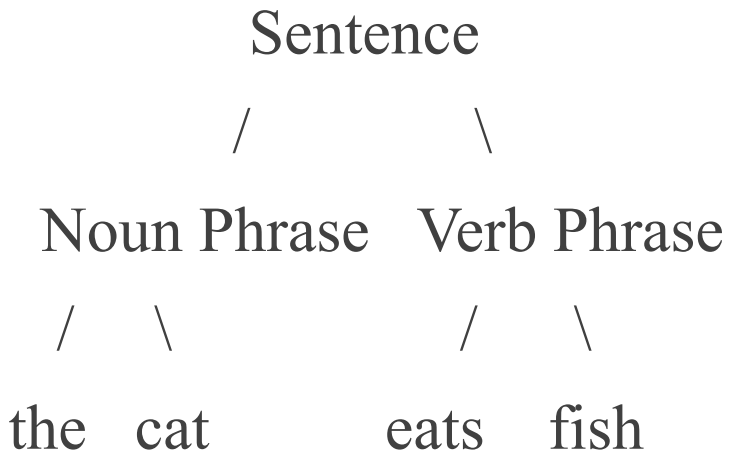
Word	Type
the	Determiner
cat	Noun
eats	Verb
fish	Noun

Grammar NLP

3. **Parsing the Sentence**

For example:
“The cat eats fish”

Computer uses the rules and dictionary to **build a tree:**



Grammar NLP

Limitations

- If grammar was wrong or incomplete → parsing failed

  "Dog the chased cat the."

- Could not handle sentences like:

  "I saw a man with binoculus." (ambiguous)

Statistical NLP

It’s an approach to language where we **let data, statistics, and probabilities** guide how a machine understands language, instead of relying only on hard-coded rules.

Techniques of Statistical NLP :

- N-grams
- HMM

Statistical NLP

a. **N-grams**

- A **chunk (or sequence)** of **N words** taken **together** from a piece of text.
- This technique captures how words occur together and allows a computer to learn the structure and patterns of language from actual text.

- **1-gram (unigram):** Single word.
  - e.g., ["I", "want", "to", "book", "a", "flight"]
- **2-gram (bigram):** Pair of words.
  - e.g., ["I want", "want to", "to book", "book a", "a flight"]
- **3-gram (trigram):** Triplet of words.
  - e.g., ["I want to", "want to book", "to book a", "book a flight"]

To **predict the next word**:

If you type: **"I want to book a"**,

N-gram can say:

- P(flight|"I want to book a") = 0.9
- P(room|"I want to book a") = 0.1

Result:

✅ Suggest **"flight"** as the next word.

Statistical NLP

b. Hidden Markov Models (HMM)

Sentence:

> "I want to book a flight"

Step 1. States (Hidden):

Possibly: PRONOUN (I), VERB (want), TO (to), VERB (book), DET (a), NOUN (flight)

Step 2. Model This As a Chain:

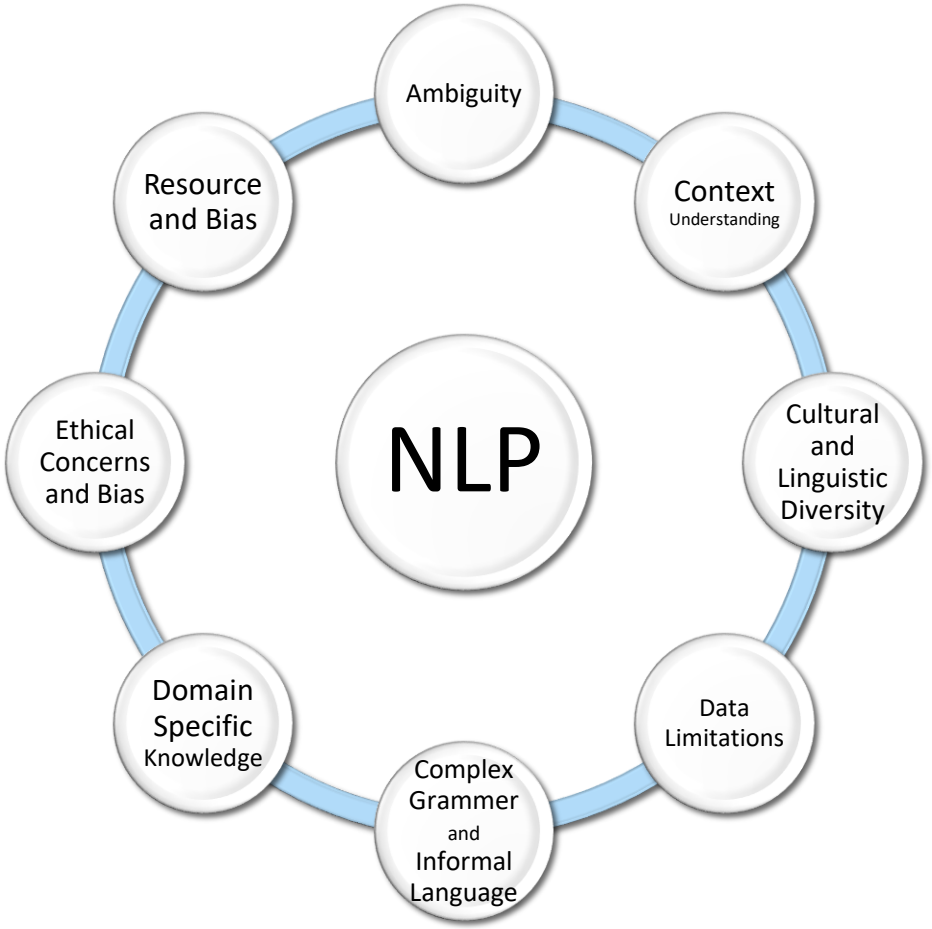
Words: **I → want → to → book → a → flight**

Tags: PRN V TO V DET N

HMM computes:

- The best tag sequence = the one with the **highest joint probability**.

Challenges in NLP



Ambiguity

Ambiguity: a word or sentence has multiple possible meanings.

Examples:

Word : Bank => river, financial

Sentence : I saw the dog with binoculus,

The **bat** flew out of the cave,

Context Understanding

Context understanding is the ability of an NLP system to **understand the intended meaning of words, phrases, or sentences based on the surrounding words, situation, and background knowledge**.

Examples:

- ❑ **“Oh great, another rainy day.”**
- It may sound like the speaker is happy, but **actually means the opposite** — they're **complaining**.
- Humans get it from **tone or past knowledge**.
- But the word "great" looks **positive**, so NLP model may misclassify it as **happy**.

Context Understanding

- Arjun met Rohan at the park. **He** gave him a gift.

  Who gave the gift to whom?

  He = Arjun? Or **Rohan**?

Cultural and Linguistic Diversity

Human languages are **not standardized** like programming languages. People speak in:

Different **dialects**, **accents**, **slangs**

Varying **word order** or **formality levels**

Deep **cultural idioms or references**

One model trained on **US English** may not work on **Indian English**.

Regional idioms can’t be translated literally.

Complex Grammar and Informal Language

While many NLP techniques depend on grammars, real-world users:

Use **incomplete sentences**

Mix **slang, emojis, abbreviations**

Skip punctuation or misspell words

Examples:

“u gonna come or nah?”

“idk, lol that’s crazy 😭 😭 ”

Domain-Specific Knowledge

Each domain (medicine, law, finance, engineering) has:

- Technical jargon
- Abbreviations
- Special grammar styles

Examples:

In medicine: “CABG” = Coronary Artery Bypass Grafting

In law: “Prima facie” has a very specific legal meaning

A model trained on Wikipedia won’t understand **hospital discharge notes**.

Ethical Concerns and Bias

NLP systems learn from the data they are trained on. If that data includes biased language, stereotypes, or unequal representation — **the model will reflect and amplify those biases.**

Examples:

Gender bias: “Man is to computer programmer as woman is to homemaker.”

Racism, religious bias, or cultural stereotypes in training corpora

NLP Problem:

Difficult to detect and correct these biases automatically

Can lead to **unfair outcomes** in chatbots, hiring tools, or legal text analysis

Resource Intensity (Computational Cost)

- Modern NLP systems are expensive and heavy to run.

- To train large language models like GPT or BERT, you need:
- Enormous GPU power (e.g., 1000+ GPUs)
- Huge storage and memory (for billion-parameter models)

- Only big tech companies can afford such resources.
- Even deploying such models (inference) is expensive.

Regular expressions

- Regular expressions (Regex) are sequences of characters used to define search patterns in text.
- They are a powerful tool for text matching, extracting, and manipulating data.

Regular Expressions used :

- To search for specific patterns in text.
- To extract or replace parts of text.
- To validate input formats, like emails, phone numbers, or postal codes

- In regex (regular expressions), we use **symbols** to build patterns.
- Some of those symbols have **special meanings** — they are **not just letters**.
- These special symbols are called **metacharacters**.

1. **.(Dot)**

   Matches: Any single character except newline (\n)

[71]:

```
import re
print(re.findall(r"a.b", "acb a_b a1b a\nb"))
```

```
['acb', 'a_b', 'a1b']
```

2. ^ (Caret)

Matches: Start of a string

[85]:

```
text = """Hello world.
This is a test.
Another Hello later."""

pattern = r"^Hello"

matches = re.findall(pattern, text)
print(matches)
```

```
['Hello']
```

3. $ (Dollar Sign)

Matches: End of a string

•[99]:

```
text = """This is the text where Hello is at last"""

pattern = r"last$"

matches = re.findall(pattern, text)
print(matches)
```

```
['last']
```

4. * (Asterisk)

Matches: 0 or more repetitions of the previous character

```
import re

text = "He said: h ho hooo!"
pattern = r"ho*"

matches = re.findall(pattern, text)
print(matches)
```

```
['h', 'ho', 'hooo']
```

5. + (Plus)

Matches: 1 or more repetitions of the previous character

[117]:

```
import re

text = "He said: h ho hooo!"
pattern = r"ho+"

matches = re.findall(pattern, text)
print(matches)
```

```
['ho', 'hooo']
```

6. ? (Question Mark)

Matches: 0 or 1 occurrence of the previous character

```
pattern = r"colou?r"
text = '''color, colour colouuur'''

matches = re.findall(pattern, text)

print(matches)
```

```
['color', 'colour']
```

7. [] (Square Brackets)

Matches: Any one character listed inside

```
pattern1 = r"[aeiou]"
pattern2 = r"[0-9]"

text = '''Hi Everyone, we are 3rd year AI at ADCKKD.
          Contact : 7794925663 for any help !!!'''

print("-----------Pattern 1-----------" )
matches1 = re.findall(pattern1, text)
matches2 = re.findall(pattern2, text)


print(matches1,end='\n\n')
print("-----------Pattern 2-----------" )

print(matches2)
```

8. [^] (Caret inside Brackets)

Matches: Any one character **NOT** in the list

```
pattern1 = r"[^aeiou]"
pattern2 = r"[^0-9]"

text = '''Hi Everyone, we are 3rd year AI at ADCKKD.
          Contact : 7794925663 for any help !!!'''

print("-----------Pattern 1-----------" )
matches1 = re.findall(pattern1, text)
matches2 = re.findall(pattern2, text)


print(matches1,end='\n\n')
print("-----------Pattern 2-----------" )

print(matches2)
```

9. {n} {n,} {n,m}

Matches: Exact or range repetitions

```
pattern1 = r"a{3}"
# Matches: aaa

pattern2 = r"a{2,}"
# Matches: aa, aaa, aaaa...

pattern3 = r"a{2,4}"
# Matches: aa, aaa, aaaa (but not aaaaa)

text = """a aa allu arjun """

matches = re.findall(pattern1, text)
print(matches)
matches = re.findall(pattern2, text)
print(matches)
matches = re.findall(pattern3, text)
print(matches)
```

10. | (Pipe)

Means: Logical OR

```
pattern = r"Ramesh|Suresh|dasu"
text = '''Ramesh, Suresh & Vasu, dasu are best friends. '''

matches = re.findall(pattern, text)

print(matches)
```

```
['Ramesh', 'Suresh', 'dasu']
```

11. () (Parenthesis)

Groups expressions or captures matched text

```
pattern1 = r"(ab)+"
pattern2 = r"(cat|dog)s" #first it search for cat or dog and checks s after to it.
text1 = """ab abab ababab"""

matches = re.findall(pattern1, text1)
print(matches)

text = "cats dogs catsanddogs cat dog catsdog"
matches = re.findall(pattern2, text)
print(matches)
```

```
['ab', 'ab', 'ab']
['cat', 'dog', 'cat', 'dog', 'cat']
```

12. \ (Backslash)

Escapes special characters, or uses special codes

[255]:
```
dot_pattern = r"\."
# Matches a literal dot

dig_pattern = r"\d"   # Matches any digit [0-9]
n_dig_pattern = r"\D"   # Matches any non-digit

word_pattern = r"\w"   # Matches any word character (a-z, A-Z, 0-9, _)
n_word_pattern = r"\W"   # Matches any non-word character

whiteSpace_pattern = r"\s"   # Matches any whitespace (space, tab, newline)
non_whiteSpace_pattern = r"\S"   # Matches any non-whitespace

text = """Hi Hello Namaste, contact me at 77xxxxx663.. !!!"""
matches = re.findall(dot_pattern, text)
print("dot_pattern =>",matches,end='\n\n')

matches = re.findall(dig_pattern, text)
print("dig_pattern =>",matches,end='\n\n')

matches = re.findall(n_dig_pattern, text)
print("n_dig_pattern =>",matches,end='\n\n')

matches = re.findall(word_pattern, text)
print("word_pattern =>" ,matches,end='\n\n')

matches = re.findall(n_word_pattern, text)
print("n_word_pattern =>",matches,end='\n\n')

matches = re.findall(whiteSpace_pattern, text)
print("whiteSpace_pattern =>",matches,end='\n\n')

matches = re.findall(non_whiteSpace_pattern, text)
print("non_whiteSpace_pattern =>",matches,end='\n\n')
```

```
dot_pattern => ['.', '.']

dig_pattern => ['7', '7', '6', '6', '3']

n_dig_pattern => ['H', 'i', ' ', 'H', 'e', 'l', 'l', 'o', ' ', 'N', 'a', 'm', 'a', 's', 't', 'e', ',', ' ', 'c', 'o', 'n', 't', 'a',
'c', 't', ' ', 'm', 'e', ' ', 'a', 't', ' ', 'x', 'x', 'x', 'x', 'x', '.', '.', ' ', '!', '!', '!']

word_pattern => ['H', 'i', 'H', 'e', 'l', 'l', 'o', 'N', 'a', 'm', 'a', 's', 't', 'e', 'c', 'o', 'n', 't', 'a', 'c', 't', 'm', 'e',
'a', 't', '7', '7', 'x', 'x', 'x', 'x', 'x', '6', '6', '3']

n_word_pattern => [' ', ' ', ',', ' ', ' ', ' ', ' ', '.', '.', ' ', '!', '!', '!']

whiteSpace_pattern => [' ', ' ', ' ', ' ', ' ', ' ', ' ']

non_whiteSpace_pattern => ['H', 'i', 'H', 'e', 'l', 'l', 'o', 'N', 'a', 'm', 'a', 's', 't', 'e', ',', 'c', 'o', 'n', 't', 'a', 'c',
't', 'm', 'e', 'a', 't', '7', '7', 'x', 'x', 'x', 'x', 'x', '6', '6', '3', '.', '.', '!', '!', '!']
```

13. \b (Word Boundary)

Matches the boundary between a **word** and a **non-word**

```
pattern = r"\bword\b"
# Matches only the exact word "word", not "sword" or "wording"
text = "Words are plural and word is singular."

matches = re.findall(pattern, text)
print(matches)
```

```
['word']
```

Do Yourself

1. Matching an Email Address
2. Matching a Phone Number
3. Validating a Postal Code
4. Detecting a Date Format (dd/mm/yyyy)

Benefits of Regular Expressions:

- Saves time in text processing tasks.
- Highly customizable for complex patterns.
- Works across many programming languages (Java, Python, JavaScript, etc.).

Finite State Automata (FSA)

- Finite State Automata (FSA) is like a small rule-following machine that **reads input step by step** and **changes its state** depending on the letter or symbol it reads.
- If, after reading all input, it ends up in a **final or accepting state**, it means the input is **valid** or **accepted**.
- FSAs are used in many real-world applications like **spell checkers, search engines, and pattern matchers.**

Finite State Automata (FSA)

Let's take a word: "cat"

- Suppose we want to build an FSA that only **accepts the word "cat"**.
- That means the machine should say "valid" only if the input is exactly c, then a, nothing more, nothing less.
- So, we define some **states**, and then set rules about how to move from one state to another.

Finite State Automata (FSA)

We define 4 states:

S0: starting state (before reading anything)

S1: after reading c

S2: after reading a

S3: after reading t (final state — if we end here, input is accepted)

Finite State Automata (FSA)

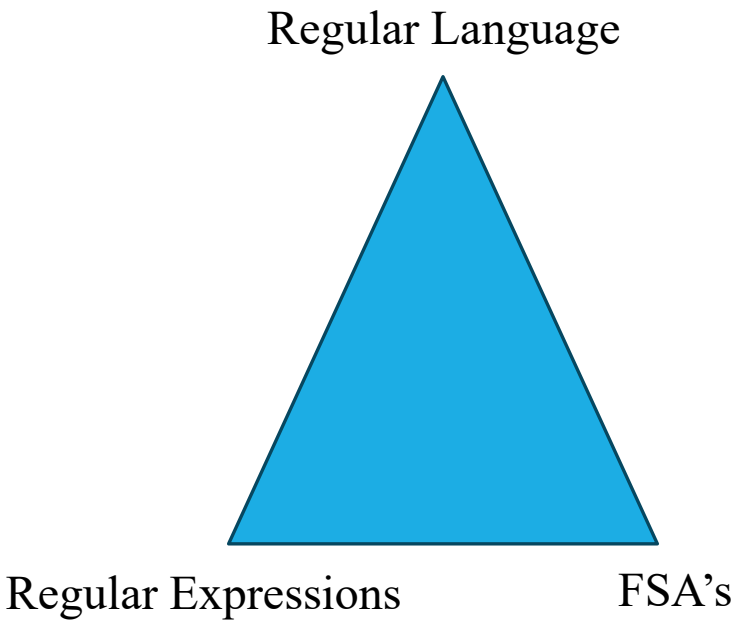
Transitions are simple rules that tell the FSA how to move. For example:

- From S0, if it reads c, it moves to S1
- From S1, if it reads a, it moves to S2
- From S2, if it reads t, it moves to S3

If the machine reaches state S3 after reading all the letters, it means the word is **accepted**. Otherwise, it's rejected.

Finite State Automata (FSA)

.



Types of FSA's

1. Deterministic Finite Automaton (DFA)
2. Nondeterministic Finite Automaton (NFA)

Deterministic Finite Automaton (DFA)

A **Deterministic Finite Automaton** is a type of FSA where for each state, there is exactly **one unique transition** for every possible input symbol.

In simple terms, the machine always knows what to do for every input—it can never be confused.

Deterministic Finite Automaton (DFA)

To illustrate this, consider a DFA designed to accept the exact word **"understand"**.

The DFA would have a chain of 10 states, each corresponding to one letter of the word. It starts at the initial state q0, reads 'u' and moves to q1, then reads 'n' and moves to q2, and so on until it finally reads 'd' and enters the accepting state q10.

q0 -u→ q1 -n→ q2 -d→ q3 -e→ q4 -r→ q5 -s→ q6 -t→ q7 -a→ q8 -n→ q9 -d→ q10

Deterministic Finite Automaton (DFA)

If the input is anything other than “understand” like:

"under",

"understanding", or

"understanded“

the **DFA will reject** it because the input does not follow the exact required path. This strict structure makes DFA ideal for **recognizing fixed patterns or exact sequences.**

Nondeterministic Finite Automaton

2. Nondeterministic Finite Automaton

- In contrast, a **Nondeterministic Finite Automaton** allows **multiple transitions** for the same input symbol and may also include paths where no transition is defined for some symbols.
- This makes NFAs more flexible and often simpler to design for complex patterns

Nondeterministic Finite Automaton

Let’s say we want an NFA to accept any word that **contains the substring "stand"**, which appears in "understand", "standalone", and "upstanding".

In this NFA, the machine may loop through any number of characters in the initial state (q0) until it detects 's', then moves through states that expect 't', 'a', 'n', and finally 'd'. Once it reaches the state after 'd', it accepts the word.

q0 -(any letters)→ q0
q0 -s→ q1 -t→ q2 -a→ q3 -n→ q4 -d→ [q5]

Nondeterministic Finite Automaton

So, "understand" would be accepted since it includes "stand" in the middle, and so would other words like "withstand" or "understandable".

If the word does not contain the substring "stand", the NFA would reject it. Unlike DFA, NFA **can explore many paths at once**, making them powerful for pattern matching.

Morphological parsing

- Morphological parsing is the **process** of analyzing a word to **identify its morphemes** (the smallest meaning-bearing units) and to extract the base form (called *lexical form*) along with grammatical features like tense, plurality, or case.
- In simpler terms, it helps break down words like **"merging"** into **merge (verb root)** + **present participle (-ing)**.
- It's highly useful in **machine translation**, **spell checking**, **lemmatization**, and **information extraction**.

Applications:

Stemming: Cutting a word to its root → *“fishing”* → *“fish”*

Lemmatization: Getting dictionary base → *“better”* → *“good”*

Morphological tagging: Analyzing word parts → *“unfriendly”* → *[“un-”, “friend”, “-ly”]*

Chatbots (e.g., Siri, Alexa)

Search engines (Google corrects and expands your queries)

Spell check and grammar tools (like Grammarly)

English Morphology

- **English Morphology** is the branch of linguistics that studies the structure and formation of words in the English language.
- It focuses on how morphemes (the smallest units of meaning) are combined to form words.

Morpheme:

A morpheme is the smallest meaningful unit in a language.

•Example: In the word **"dogs"**, there are two morphemes:

- •*dog* (root word)
- •*-s* (plural suffix)

Types of Morphemes:

Free Morphemes: Can stand alone as words

e.g., *book*, *run*, *fast*

Bound Morphemes: Cannot stand alone, must be attached to other morphemes

e.g., *-ed*, *-s*, *un-*, *-ing*

Terminology in Morpheme

Stem: The core part of a word that carries meaning.
E.g., *"run"* in *"running"*

Affixes: Units added to a stem to alter its meaning or role.
These can be:

- **Prefix** (before the stem): *un-* in *unhappy*
- **Suffix** (after the stem): *-s* in *books*

Example breakdown:

"unbelievable" → un- (prefix) + believe (stem) + -able (suffix)

Types of Morphological Formation

English uses **concatenative morphology**, which means stems and affixes are strung together (concatenated) to build words.

There are 4 major word-formation processes:

- **Inflection**:

Changes the form of a word to express **tense, number, possession, comparison**, etc., **without** creating a new word.

Inflection

Base Word	Inflected Form
walk	
walk	
walk	
cat	
boy	
big	
big	
run	
study	

Inflection

Base Word	Inflected Form
walk	walks
walk	
walk	
cat	
boy	
big	
big	
run	
study	

Inflection

Base Word	Inflected Form
walk	walks
walk	walked
walk	
cat	
boy	
big	
big	
run	
study	

Inflection

Base Word	Inflected Form
walk	walks
walk	walked
walk	walking
cat	
boy	
big	
big	
run	
study	

Inflection

Base Word	Inflected Form
walk	walks
walk	walked
walk	walking
cat	cats
boy	
big	
big	
run	
study	

Inflection

Base Word	Inflected Form
walk	walks
walk	walked
walk	walking
cat	cats
boy	boy’s
big	
big	
run	
study	

Inflection

Base Word	Inflected Form
walk	walks
walk	walked
walk	walking
cat	cats
boy	boy’s
big	bigger
big	
run	
study	

Inflection

Base Word	Inflected Form
walk	walks
walk	walked
walk	walking
cat	cats
boy	boy’s
big	bigger
big	biggest
run	
study	

Inflection

Base Word	Inflected Form
walk	walks
walk	walked
walk	walking
cat	cats
boy	boy’s
big	bigger
big	biggest
run	runs
study	

Inflection

Base Word	Inflected Form
walk	walks
walk	walked
walk	walking
cat	cats
boy	boy’s
big	bigger
big	biggest
run	runs
study	studying

Derivation

Adding **prefixes** or **suffixes** to a **base word to create a new word**.

Base Word	New Word
happy	
child	
hope	
beauty	
care	
read	
act	
appear	
friend	
help	

Compounding

Joins two stems to form a new word. **Compounds** are either closed (e.g., football), hyphenated (e.g., mother-in-law), or open (e.g., post office).

Compound Word	Components
toothbrush	tooth + brush
blackboard	black + board
bookstore	book + store
bedroom	bed + room
laptop	lap + top
toothpaste	tooth + paste
snowman	snow + man
playground	play + ground
football	foot + ball
wallpaper	wall + paper

Conversion (Zero Derivation)

Changing the **word class** without changing the word form.

Original Word (POS)	Converted Word (POS)
email (noun)	to email (verb)
run (verb)	a run (noun)
text (noun)	to text (verb)
clean (adj)	to clean (verb)
bottle (noun)	to bottle (verb)
Google (noun)	to google (verb)
record (noun)	to record (verb)
kick (verb)	a kick (noun)
brush (noun)	to brush (verb)
host (noun)	to host (verb)

Clipping

Shortening a longer word **without changing its meaning**.

Clipped Word	Full Word
exam	examination
phone	telephone
gym	gymnasium
lab	laboratory
fridge	refrigerator
math	mathematics
ad	advertisement
vet	veterinarian
bus	omnibus
demo	demonstration

Blending

Merging parts of two words to make a new word.

Blend	Source Words
brunch	breakfast + lunch
smog	smoke + fog
motel	motor + hotel
podcast	iPod + broadcast
hangry	hungry + angry
webinar	web + seminar
spork	spoon + fork
blog	web + log
Brexit	Britain + exit
infotainment	information + entertainment

Parsing English Morphological output

.

Input	Morphological parsed output
cats	cat +N +PL
cat	cat +N +SG
cities	city +N +PL
geese	goose +N +PL
goose	(goose +N +SG) or (goose +V)
gooses	goose +V +3SG
merging	merge +V +PRES-PART
caught	(caught +V +PAST-PART) or (catch +V +PAST)

Transducers for Lexicon and Rules

Imagine a machine that takes a word and tells you everything about it—what it means, its base form, its tense, and so on. That's what a **transducer** does in language processing.

Technically, a **transducer** is a special kind of **Finite State Machine (FSM)** that reads **input symbols** and produces **output symbols**.

So, instead of just checking whether a word is valid (like a simple automaton), it actually translates it from one form to another.

For example, if the input is "cats", a transducer might output "cat +NOUN +PLURAL".

Languages are full of **changes and variations**. Words can appear in many different forms: "walk", "walked", "walking", "walks".

These are all **surface forms** that humans understand easily. But for a computer, each form looks completely different unless we teach it the connection.

That's where transducers come in—they help map the **surface forms** (the actual word we see) to their **lexical forms** (the base word and grammatical information).

- A **transducer** in morphology is a **machine (or a program)** that **maps between two forms**:

1. Lexical form walk + PAST
2. Surface form walked

- A **Finite-State Transducer (FST)** can take a **lexical entry** and apply **morphological rules** to produce the surface word.
- It's also **bidirectional** i.e., it can also take "walked" and output "walk + PAST".

1. Lexicon (Dictionary)

Lemma	IPA	POS	Syntax Type	Morphology	Meaning
cat	/kæt/	Noun	Countable	cats	A small domestic animal
run	/rʌn/	Verb	Intransitive	runs, ran, running	To move quickly by foot
write	/raɪt/	Verb	Transitive	writes, wrote, writing, written	To form letters with a pen
happy	/ˈhæpi/	Adjective	-	happier, happiest	Feeling or showing pleasure
box	/bɒks/	Noun	Countable	boxes	A container with flat surfaces
drive	/draɪv/	Verb	Transitive	drives, drove, driven, driving	Operate and control a vehicle
good	/gʊd/	Adjective	-	better, best (irregular)	Of high quality
child	/ʧaɪld/	Noun	Countable	children (irregular plural)	A young human
study	/ˈstʌdi/	Verb	Transitive	studies, studied, studying	Learn about a subject
eat	/iːt/	Verb	Transitive	eats, ate, eaten, eating	To consume food

1. Lexicon (Dictionary)

Lemma	POS	Morphology
cat	Noun	cats
run	Verb	runs, ran, running
write	Verb	writes, wrote, writing, written
happy	Adjective	happier, happiest
box	Noun	boxes
drive	Verb	drives, drove, driven, driving
good	Adjective	better, best (irregular)
child	Noun	children (irregular plural)
study	Verb	studies, studied, studying
eat	Verb	eats, ate, eaten, eating

2. Morphological Rules

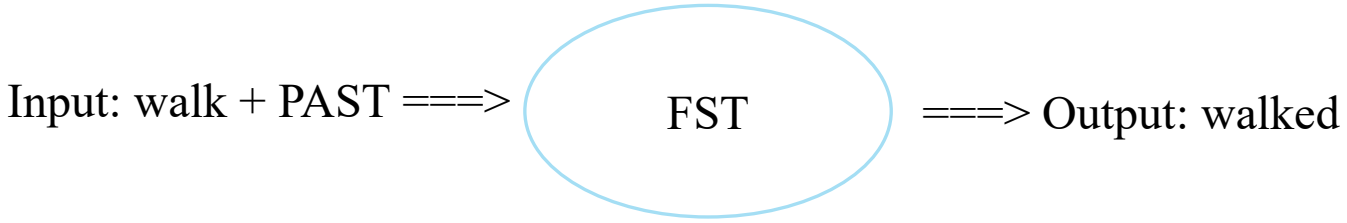
These are the **transformations** we apply to those base words using affixes (prefixes/suffixes) or stem changes.

Let have some common morphological changes in English and show how a transducer applies rules.

Rule 1: Past Tense (Regular Verbs)

Lexical: walk + PAST

Rule: If it's a regular verb, add "ed"



Inflection

1. Plural Formation (Nouns)

- Regular:
  - cat + **PL** → cats
  - dog + **PL** → dogs
- Rules:
  - Add **–s** to most nouns
  - Add **–es** after **s, x, z, ch, sh**: box → boxes
  - If ends in **consonant + y**, change **y** → **ies**: baby → babies
  - If ends in **f/fe**, change → **ves**: knife → knives
- Irregular:
  - child → children, man → men, mouse → mice

2. Tense/Aspect (Verbs)

- **Past Tense**:
  - walk + PAST → walked
  - If ends in **e**, add **–d**: love → loved
  - If ends in consonant + y, change y → ied: try → tried
- Irregulars:
  - go → went, run → ran, eat → ate, cut → cut
- **Progressive Aspect (Present Participle)**:
  - run + PROG → running
  - If ends in e, drop it: make → making
  - Double consonant after short vowel: sit → sitting

3. 3rd Person Singular (Present)

- Add **–s**: play → plays
- After ch, sh, s, x, add **–es**: watch → watches
- carry → carries

4. Comparatives and Superlatives (Adjectives)

- big → bigger → biggest
- happy → happier → happiest
- For longer adjectives: beautiful → more beautiful → most beautiful

II. DERIVATIONAL MORPHOLOGICAL RULES

These rules form **new words** (new meaning or new class).

1. Prefixation (add to beginning)

- un–: happy → unhappy
- re–: do → redo
- dis–: agree → disagree
- pre–: view → preview
- in–: visible → invisible

2. Suffixation (add to end)

a. Noun-forming

- –er: teach → teacher
- –ness: kind → kindness
- –tion: inform → information
- –ity: real → reality

b. Verb-forming

- –ize: modern → modernize
- –en: strength → strengthen

c. Adjective-forming

- –ful: beauty → beautiful
- –less: hope → hopeless
- –able: read → readable

d. Adverb-forming

- –ly: quick → quickly, slow → slowly

1. Conversion (Zero Derivation)

- No affix, only class change:
  - Google (noun) → to Google (verb)
  - text → to text, run → a run

2. Clipping

- Shortening a longer word:
  - advertisement → ad
  - laboratory → lab
  - influenza → flu

4. Reduplication

- Duplicate all or part of a word (rare in English):
  - bye-bye, go-go, boo-boo

5. Acronym & Initialism

- **Acronym**: NASA (National Aeronautics and Space Administration)

IV. INFIXATION (RARE in English)

- Inserting inside a word:
  - Rare and informal: fan-bloody-tastic, un-freaking-believable

V. CIRCUMFIXATION (Not in English)

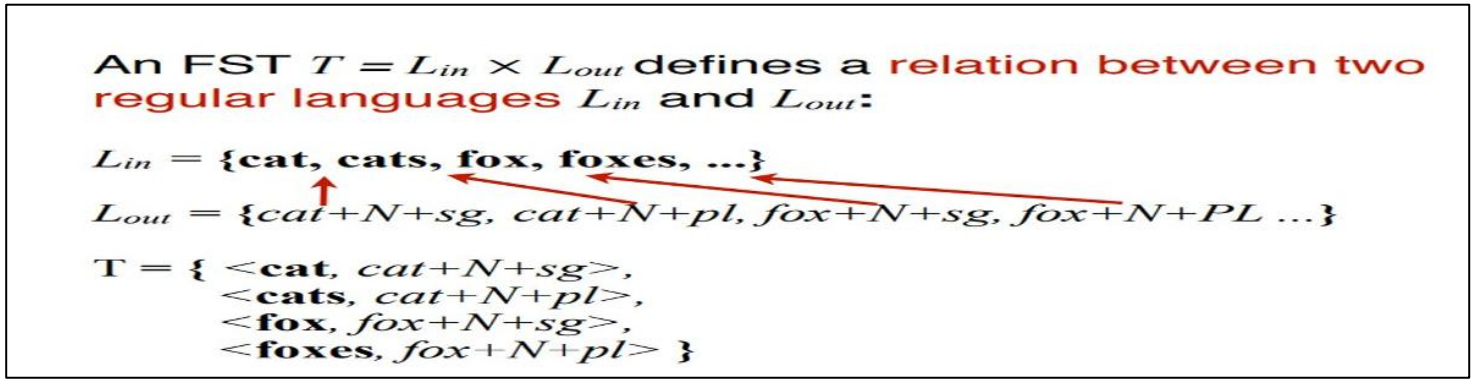
- Found in **German, Dutch**, etc.
- Example (German):
  - geliebt = ge (prefix) + lieb (root) + t (suffix) → "loved"

A **Finite State Transducer (FST)** combines the lexicon and rules to either:

Analyze a word (e.g., "played" → "play +V +PAST")

Generate a word (e.g., "play +V +PAST" → "played")

It does this by reading input characters one by one, moving between **states**, and generating corresponding output symbols.



Let’s walk through how a transducer processes the word "played":

In Analysis:

- Input: "played"
- The transducer identifies "play" as the root verb.
- It sees "ed" as a past tense suffix.
- Output: "play +V +PAST"

In Generation:

- Input: "play +V +PAST"
- The transducer knows the rule: "add -ed to verb for past tense"
- Output: "played"

So, the same machine can be used **in both directions** — to **analyze** or to **generate** words.

Transducers are powerful and used in many **language-based technologies**:

- **Spell checkers**: To recognize root forms of misspelled words.
- **Machine translation**: To convert between languages by identifying root meanings.
- **Speech synthesis**: To convert text to speech, you need to know how words are built.
- **Search engines**: So they can understand queries like "running" and find documents with "run".

TOKENSIZATION

Computers don’t think like humans, they don’t understand words or grammar the way we do. To a computer, a sentence is just a **long string of characters** like this:

Sentence: "What restaurants are nearby?"

- Unless we split (tokenize) this, the computer cannot:
  - Recognize **words** (like “What", “restaurants", “are“, “nearby”)
  - Know **what each part means**
  - Match these words with its vocabulary or training data

Tokenization

- Tokenization is a process in **Natural Language Processing (NLP)** where we take a large block of text (like a sentence or paragraph) and break it down into smaller units called **tokens**.
- These tokens are usually **words**, **characters**, or **subwords**.
- This step is very important because machines don't understand human language naturally, so we split it into parts that are easier for them to process and analyze.
- For example:
  Input sentence: "What restaurants are nearby?"
  After tokenization: ['What', 'restaurants', 'are', 'nearby']
-

This helps the computer understand and work with each word individually.

Types of Tokenization

There are **three major types** of tokenization used in NLP, and each has its own purpose and benefits:

1. word tokenization

Word tokenization is the most common and simple form. It breaks the text into **individual words** using spaces and punctuation marks as **delimiters**.

Example:
Text: “How are you?” → Tokens: ['How', 'are', 'you']

Issue: If the model sees a **rare or unknown word** (called an **OOV – Out Of Vocabulary** word), it may not recognize it. In that case, the unknown word is replaced by a generic symbol like [UNK] (unknown).

With Word Tokenization, The tokenizer will **split the sentence into words**:

- ['The', 'child', 'is', 'giggling', 'loudly']

Now, since **"giggling"** is **OOV (out of vocabulary)**, the model can't recognize it. So it replaces it with a special placeholder token, usually [UNK]:

- ['the', 'child', 'is', '[UNK]', 'loudly']

❖ Replacing with [UNK] may **reduce accuracy**, because:

- Context understanding becomes weaker.

2. Character tokenization

This method splits a sentence into **individual characters** instead of words.

Example:
Text: “Hello” → Tokens: ['H', 'e', 'l', 'l', 'o']

It solves the OOV problem because every word is just a combination of characters, so even unknown words can be understood at the character level.

3. Subword tokenization (Morphological tokenisation)

Subword tokenization is a smart middle-ground between word and character tokenization. It breaks words into **meaningful parts** (called subwords), like **prefixes, suffixes**, or **roots**.

Example:
Sentence: “What is the tallest building?”
Tokens: ['what', 'is', 'the', 'tall', 'est', 'build', 'ing']

This method helps when a rare word like **“machinating”** appears. Even if the model doesn't know the full word, it can split it into known subwords like ['machin', 'ating'], or ['machinate', 'ing'].
Now, it understands that **‘ing’** is a common verb suffix and can **guess the role** of the word in the sentence — for example, as a verb in action or as a noun.

Error Detection and Correction Spelling Errors

- Spelling error detection and correction is a task in NLP where a machine identifies mistakes in written text (like typos or misspellings) and suggests or applies the correct version of the word.
- This is an essential step in systems like chatbots, search engines, autocorrect tools, and grammar checkers.
- The goal is to **maintain the meaning of the sentence** while fixing the misspelled parts.

There are mainly **two types** of spelling errors:

a) Non-word errors

These occur when a word doesn't exist in the dictionary.
Example: "recieve" → should be "receive"

b) Real-word errors

These happen when the word is spelled correctly but **used incorrectly** in context.
Example: "Their going to the park" → should be "They're going to the park"

Non-word errors are easier to detect because they don't match any known word. Real-word errors are harder because they require context understanding.

Types of Spelling Errors

1. Insertion Errors:

- Extra letters are added.
- Example: "progrramming" → "programming“

2. Deletion Errors:

- Letters are missing.
- Example: "prgramming" → "programming"

3. Substitution Errors:

One letter is replaced by another.

Example: "progrsmming" → "programming"

4. Transposition Errors:

Adjacent letters are swapped.

Example: "progarming" → "programming"

Detecting Spelling Errors

Spelling error detection usually involves checking each word against a **lexicon (dictionary)** of correct words.

- If a word is **not found**, it's marked as a possible error (Non-word error).
- If it's a real word, **contextual models** (like Language Models) are used to check if the word makes sense in the sentence.

For example:
Input: "She has went to the store."
Detection: "went" is a valid word, but grammatically wrong after "has". This is a real-word error.

Detecting Spelling Errors

Once an error is detected, correction is about suggesting the most **likely correct word**. Here are the methods:

a) Edit Distance (Levenshtein Distance)

This measures how many changes (insertions, deletions, or substitutions) are needed to turn the wrong word into a correct one.

Example:
"realy" → to "really" (only 1 letter added)

b) Candidate Generation

The system generates a list of possible correct words based on closeness (via edit distance or phonetics).
Example for “wierd” → ['weird', 'wired', 'ward']

c) Contextual Ranking

A language model (like BERT or n-gram) ranks which suggestion fits best in the sentence.

Example:
“She is wierd.”
Candidates: “weird”, “wired”
→ “She is weird” has higher probability → selected.

Minimum Edit Distance

- **Minimum Edit Distance** is a way of measuring **how different two strings are**, by counting the **minimum number of operations** needed to convert one string into another.
- This is widely used in **spelling correction**, **plagiarism detection**, and many other areas.
- In NLP, it helps in identifying the **closest correct word** for a misspelled word.

There are typically **three basic operations** allowed to transform one word into another:

Insertion – Adding a character
Example: "recieve" → insert 'e' → "receive"

Deletion – Removing a character
Example: "adres" → delete 's' → "adre" → then insert 's' → "adres"

Substitution – Replacing one character with another
Example: "teh" → substitute 'e' with 'h' → "the"

In some cases, **transposition** (swapping two adjacent characters) is also considered, especially in spelling correction (like "freind" → "friend").

Each operation is given a **cost**, and the goal is to find the sequence of operations with the **least total cost**.

- The most common form of Minimum Edit Distance is the **Levenshtein Distance**. It allows **Insertions, Deletions**, **Substitutions**
- Each of these operations typically has a cost of 1.
- **Example:**

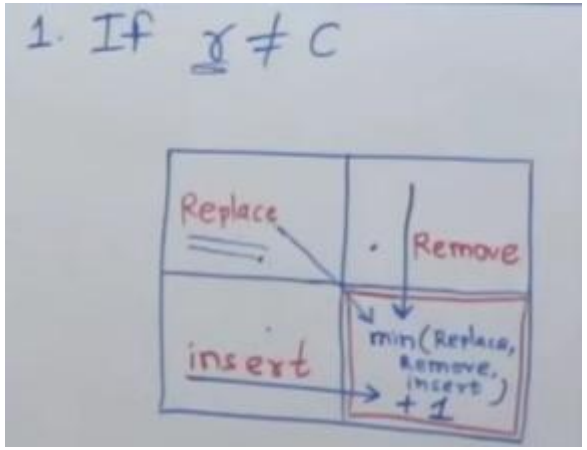
To convert **"kitten" → "sitting"**:

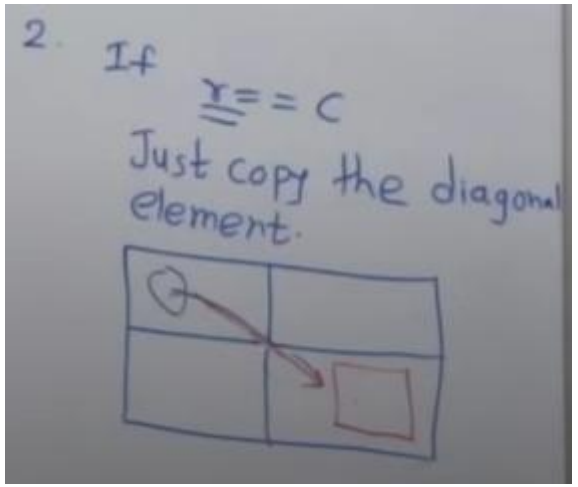
Operation	Result	Cost
Substitute 'k'→'s'	"sitten"	1
Substitute 'e'→'i'	"sittin"	1
Insert 'g'	"sitting"	1
Total Distance		**3**

Dynamic Programming Approach

- To compute MED efficiently, we use **dynamic programming** to build a **matrix (table)** where:
- Rows represent characters of the first word.
- Columns represent characters of the second word.
- Each cell [i][j] stores the cost of converting the first i letters of word1 to the first j letters of word2.
- This matrix is filled row by row using a formula that picks the **minimum cost** among insert, delete, or substitute operations.

Rules for MED





MED Matrix


NULL a b c f g
NULL 0 1 2 3 4 5
a 1 0 1 2 3 4
d 2 1 1 2 3 4
c 3 2 2 1 2 3
e 4 3 3 2 2 3
g 5 4 4 3 3 2

Applications in Spelling Correction

- Suppose a user types a misspelled word like **"recieve"**.
- The spell-check system:
- Compares it with all known dictionary words (like "receive", "relieve", "recipe", etc.)
- Computes the MED between "recieve" and each known word.
- Selects the one with the **lowest distance**.
- In this case, "receive" would likely have the **smallest MED = 1**, and thus be chosen.