NLP

NLP stands for Natural Language Processing, which is a part of Computer Science, Human language, and Artificial Intelligence. It is the technology that is used by machines to understand, analyse, manipulate, and interpret human's languages. It helps developers to organize knowledge for performing tasks such as translation, automatic summarization, Named Entity Recognition (NER), speech recognition, relationship extraction, and topic segmentation.

Components of NLP

There are the following two components of NLP -

1. Natural Language Understanding (NLU)

NLU enables machines to understand and interpret human language by extracting metadata from content. It performs the following tasks:

- Helps analyze different aspects of language.
- Helps map the input in natural language into valid representations.

Natural Language Understanding (NLU) helps the machine to understand and analyse human language by extracting the metadata from content such as concepts, entities, keywords, emotion, relations, and semantic roles. NLU mainly used in Business applications to understand the customer's problem in both spoken and written language.

2. Natural Language Generation (NLG)

Natural Language Generation (NLG) acts as a translator that converts the computerized data into natural language representation. It mainly involves Text planning, Sentence planning, and Text Realization. NLG is a method of creating meaningful phrases and sentences (natural language) from data. It comprises three stages: text planning, sentence planning, and text realization.

- Text planning: Retrieving applicable content.
- Sentence planning: Forming meaningful phrases and setting the sentence tone.
- Text realization: Mapping sentence plans to sentence structures.

Applications of NLP

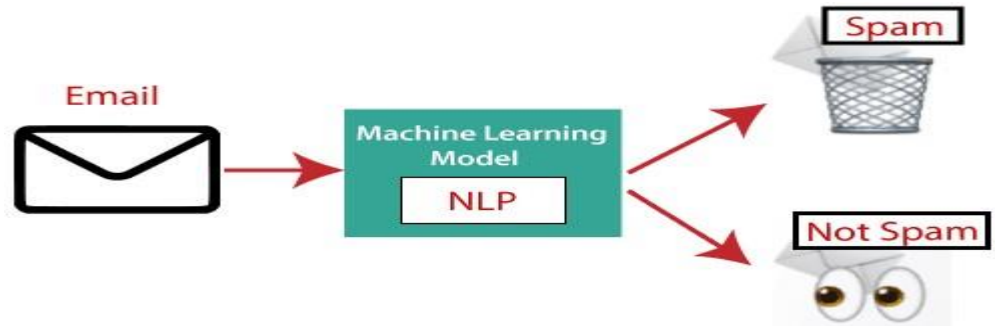
There are the following applications of NLP -

1. Question Answering

Question Answering focuses on building systems that automatically answer the questions asked by humans in a natural language.

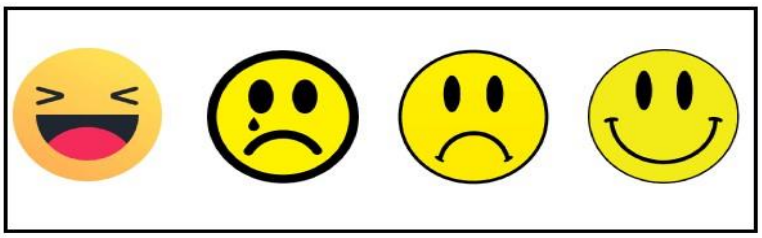
2. Spam Detection

Spam detection is used to detect unwanted e-mails getting to a user's inbox.



3. Sentiment Analysis

Sentiment Analysis is also known as **opinion mining**. It is used on the web to analyse the attitude, behaviour, and emotional state of the sender. This application is implemented through a combination of NLP (Natural Language Processing) and statistics by assigning the values to the text (positive, negative, or natural), identify the mood of the context (happy, sad, angry, etc.)



4. Machine Translation

Machine translation is used to translate text or speech from one natural language to another natural language.

Example: Google Translator

5. Spelling correction

Microsoft Corporation provides word processor software like MS-word, PowerPoint for the spelling correction.

6. Speech Recognition

Speech recognition is used for converting spoken words into text. It is used in applications, such as mobile, home automation, video recovery, dictating to Microsoft Word, voice biometrics, voice user interface, and so on.

7. Chatbot

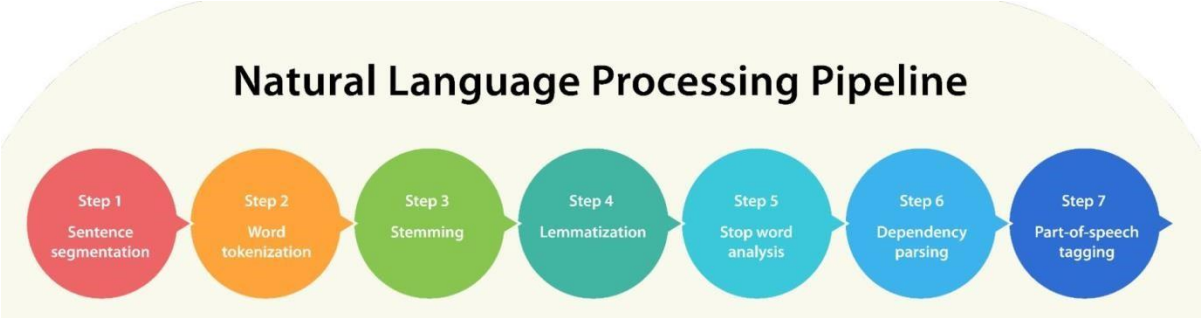
Implementing the Chatbot is one of the important applications of NLP. It is used by many companies to provide the customer's chat services.

8. Information extraction

Information extraction is one of the most important applications of NLP. It is used for extracting structured information from unstructured or semi-structured machine-readable documents.

NLP pipeline

The NLP pipeline comprises a set of steps to read and understand human language.



There are the following steps to build an NLP pipeline -

Step1: Sentence Segmentation

Sentence Segment is the first step for building the NLP pipeline. It breaks the paragraph into separate sentences.

Example: Consider the following paragraph -

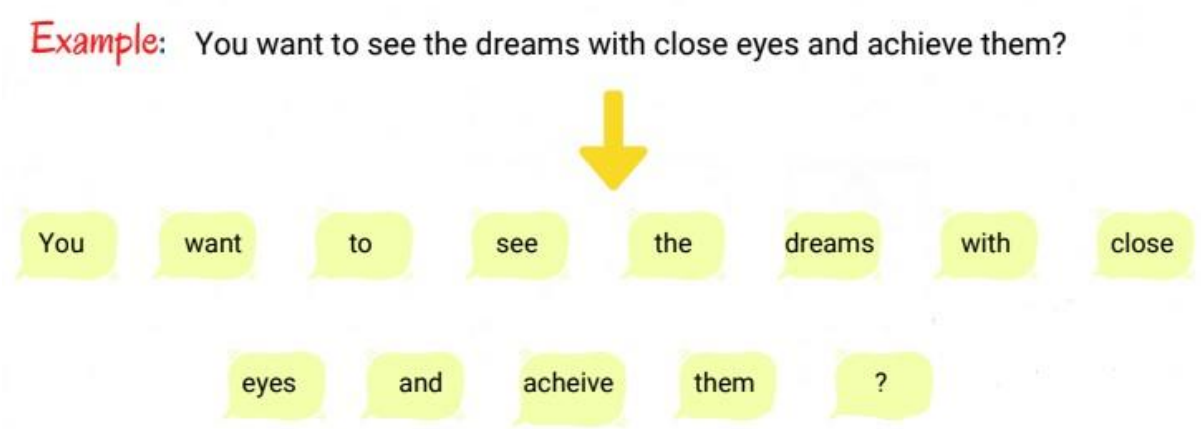
Independence Day is one of the important festivals for every Indian citizen. It is celebrated on the 15th of August each year ever since India got independence from the British rule.

Sentence Segment produces the following result:

1. "Independence Day is one of the important festivals for every Indian citizen."
2. "It is celebrated on the 15th of August each year ever since India got independence from the British rule."

Step2: Word Tokenization

Word Tokenizer is used to break the sentence into separate words or tokens.



Step3: Stemming

Stemming is a technique used to extract the base form of the words by removing affixes from them. Stemming is used to normalize words into its base form or root form. For example, celebrates, celebrated and celebrating, all these words are originated with a single root word "celebrate." The big problem with stemming is that sometimes it produces the root word which may not have any meaning.

For Example, intelligence, intelligent, and intelligently, all these words are originated with a single root word "intelligen." In English, the word "intelligen" do not have any meaning.

Example:

Words	Affixes	Stem
healing	ing	heal
dreams	s	dream
studies	es	studi

Step 4: Lemmatization

Lemmatization is quite similar to the Stamming. It is used to group different inflected forms of the word, called Lemma. The main difference between Stemming and lemmatization is that it produces the root word, which has a meaning. For example: In lemmatization, the words intelligence, intelligent, and intelligently has a root word intelligent, which has a meaning.

Words	Affixes	lemma
healing	ing	heal
dreams	s	dream
studies	es	study

Step 5: Identifying Stop Words

In English, there are a lot of words that appear very frequently like "is", "and", "the", and "a". NLP pipelines will flag these words as stop words. Stop words might be filtered out before doing any statistical analysis.

Example: He **is a** good boy.

For the purpose of analyzing text data and building NLP models, these stopwords might not add much value to the meaning of the document. Generally, the most common words used in a text are “the”, “is”, “in”, “for”, “where”, “when”, “to”, “at” etc.

Example:

Consider this text string – “There is a pen on the table”.

Now, the words “is”, “a”, “on”, and “the” add no meaning to the statement while parsing it. Whereas words like “there”, “book”, and “table” are the keywords and tell us what the statement is all about.

Step 6: Dependency Parsing

Dependency Parsing is used to find that how all the words in the sentence are related to each other.

Step 7: POS tags

POS stands for parts of speech, which includes Noun, verb, adverb, and Adjective. It indicates that how a word functions with its meaning as well as grammatically within the sentences. A word has one or more parts of speech based on the context in which it is used.

Example: **"Google"** something on the Internet.

In the above example, Google is used as a verb, although it is a proper noun.

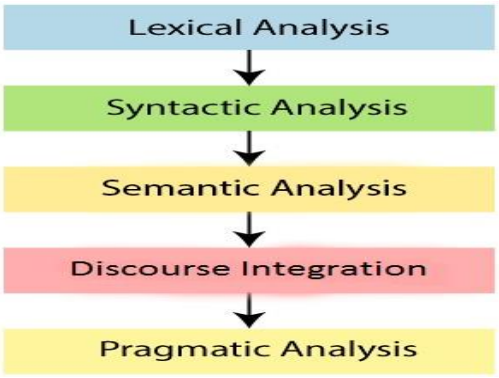
Step 8: Named Entity Recognition (NER)

Named Entity Recognition (NER) is the process of detecting the named entity such as person name, movie name, organization name, or location.

Example: **Steve Jobs** introduced iPhone at the Macworld Conference in San Francisco, California.

Phases of NLP

There are the following five phases of NLP:



1. Lexical Analysis and Morphological

The first phase of NLP is the Lexical Analysis. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. It divides the whole text into paragraphs, sentences, and words.

2. Syntactic Analysis (Parsing)

Syntactic Analysis is used to check grammar, word arrangements, and shows the relationship among the words.

Example: Agra goes to the Dilna

In the real world, Agra goes to the Dilna, does not make any sense, so this sentence is rejected by the Syntactic analyzer.

3. Semantic Analysis

Semantic analysis is concerned with the meaning representation. It mainly focuses on the literal meaning of words, phrases, and sentences.

4. Discourse Integration

Discourse Integration depends upon the sentences that proceeds it and also invokes the meaning of the sentences that follow it. It is nothing but a sense of context. That is sentence or word depends upon that sentences or words.

For example, **Ram wants it.**

In the above statement, we can clearly see that the “it” keyword does not make any sense. In fact, it is referring to anything that we don‟t know. That is nothing but this “it” word depends

upon the previous sentence which is not given. So once we get to know about “it”, we can easily find out the reference.

5. Pragmatic Analysis

Pragmatic is the fifth and last phase of NLP. It helps you to discover the intended effect by applying a set of rules that characterize cooperative dialogues. It means the study of meanings in a given language. Process of extraction of insights from the text. It includes the repetition of words, who said to whom? etc. It understands that how people communicate with each other, in which context they are talking and so many aspects.

For Example: "Open the door" is interpreted as a request instead of an order.

Why NLP is difficult?

NLP is difficult because Ambiguity and Uncertainty exist in the language.

Ambiguity

There are the following three ambiguity - ○

Lexical Ambiguity

Lexical Ambiguity exists in the presence of two or more possible meanings of the sentence within a single word.

Example:

Manya is looking for a **match**.

In the above example, the word match refers to that either Manya is looking for a partner or Manya is looking for a match. (Cricket or other match) ○ **Syntactic Ambiguity**

Syntactic Ambiguity exists in the presence of two or more possible meanings within the sentence.

The Fish is ready to eat.

Is the fish ready to eat his/her food or fish is ready for someone to eat?

- **Referential Ambiguity**

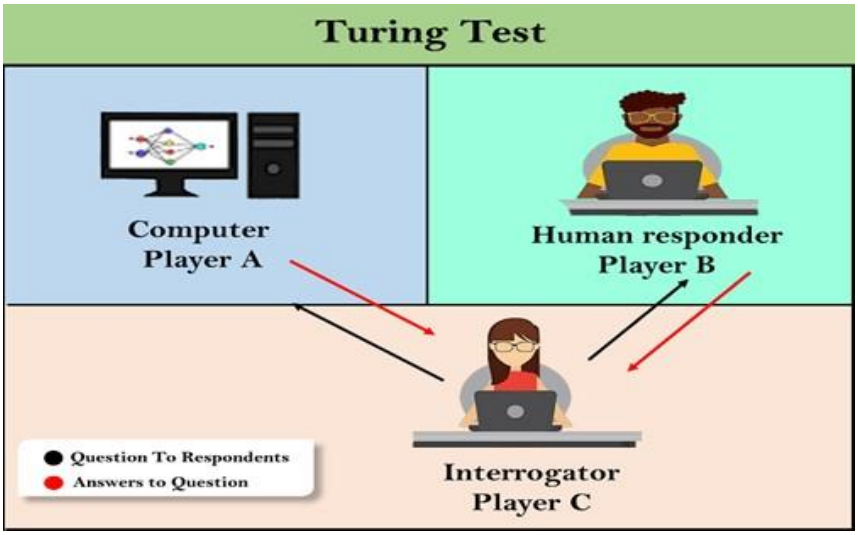
Referential Ambiguity exists when you are referring to something using the pronoun.

Example: Mia went to Sunita. She said, "I am hungry."

In the above sentence, you do not know that who is hungry, either Mia or Sunita.

Turing Test

In 1950, Alan Turing introduced a test to check whether a machine can think like a human or not, this test is known as the Turing Test. In this test, Turing proposed that the computer can be said to be an intelligent if it can mimic human response under specific conditions.



The Turing test is based on a party game "Imitation game," with some modifications. This game involves three players in which one player is Computer, another player is human responder, and the third player is a human Interrogator, who is isolated from other two players and his job is to find that which player is machine among two of them.

Consider, Player A is a computer, Player B is human, and Player C is an interrogator. Interrogator is aware that one of them is machine, but he needs to identify this on the basis of questions and their responses.

The conversation between all players is via keyboard and screen so the result would not depend on the machine's ability to convert words as speech. The test result does not depend on each correct answer, but only how closely its responses like a human

answer. The computer is permitted to do everything possible to force a wrong identification by the interrogator.

The questions and answers can be like:

Interrogator: Are you a computer?

PlayerA (Computer): No

Interrogator: Multiply two large numbers such as (256896489*456725896)

Player A: Long pause and give the wrong answer.

In this game, if an interrogator would not be able to identify which is a machine and which is human, then the computer passes the test successfully, and the machine is said to be intelligent and can think like a human.

Regular Expressions

A regular expression (a.k.a. regex or regexp) is a sequence of characters that specifies a search pattern in the text. Usually, such patterns are used by string-searching algorithms for “find” or “find and replace” operations on strings, or for input validation.

- **Sets**

Brackets [] are used to specify a disjunction of characters.

```
/[wW]oodchuck/ --> Woodchuck or woodchuck
/[abc]/ --> „a“, „b“, or „c“, It does not match abc;
/[1234567890]/ --> any digit
```

`/[a\-p]` : Matches a, -, or p. It matches - because \ escapes it;

`/[a-z0-9]` : Matches characters from a to z or from 0 to 9.

- **dash**, used to specify a range. For instance, putting A-Z in brackets allows to return all matches of an upper case letter.

```
/[A-Z]/ → machtes an upper case letter
/[a-z]/ → matches a lower case letter
/[0-9]/ → matches a single digit
```

- The **caret** ^ can be used for negation or just to mean ^.

```
/[^A-Z]/ --> not an upper case letter
/[^Ss]/ --> neither „S" nor „s"
/[^\.]/ --> not a period
/[e^]/ --> either „e" or „^"
/a^b/ --> the pattern „a^b"
```

The caret serves two different purposes.

- It is a special character that denotes “the beginning of a line” and

So **^[b-d]t$** means:

Start with b/c/d character and ending with t character. $ symbol indicate that the string should ended with character „t".

- it is a “not” operator inside of *+s.

For example: [^abc] -> not a, b or c

- The **question mark** ? marks optionality of the previous expression. For instance, putting a ? at the end of the woodchucks returns results for woodchuck (without an s) and woodchucks (with an s).

```
/woodchucks?/ --> woodchuck or woodchucks,excluding s
or including s;
/colou?r/ --> color or colour
```

- The **period** . used to specify any character between two expressions. For instance, putting beg.n will return you words such as begin or begun.

```
/beg.n/ --> Match any character between beg and n
(e.g.
begin, begun)
```

- The use of * **or** +allows you to add 1 or more of a previous character.

```
oo*h! → 0 or more of a previous character (e.g. ooh!,
ooooh!) o+h! → 1 or more of a previous character (e.g.
ooh!, oooooh!)
baa+ → baa, baaaa, baaaaaa, baaaaaaa
```

- **Character Classes**

Anchors are used to assert something about the string or the matching process.

`\w → any word character,` Matches characters a-z, A-Z, 0–9, and underscore;
`\W →anything but not a word character,` ie except a-z, A-Z, 0–9, and _
`\d → any digit character,` from 0–9;
`\D → anything but not a digit character,` Matches any non-digits
`\b → a word boundary,` Matches the word boundary (or empty string) at the start and end of a word;
`\B → anything but not a word boundary`
`\s → any space character,` include the \t, \n, \r, and space characters
`\S → anything but not a space character`

Regular Expression functions

1) strsplit(x, split)

An R Language function which is used to split the strings into substrings with split arguments.

Example 1:

```
df<-("R is the statistical analysis language") strsplit(df,
split = " ")
```

Output ="R" "is" "the" "statistical" "analysis" "language"

Split the string using " ".

Example 2: **Using strsplit() function with delimiter**

A delimiter in general is a simple symbol, character, or value that separates the words or text in the data.

```
df<-"all16i5need6is4a9long8vacation" strsplit(df,split

= "[0-9]+")
```

Output = "all" "i" "need" "is" "a" "long" "vacation"

In this example, our input has the numbers lies between 0-9. Hence we used the regular expression as [0-9]+ to split the data by removing the numbers.

2) grep() function grep(pattern, x, ignore.case = FALSE/TRUE,

value = FALSE/TRUE)

The grep allows you to "grab" the word or set of words you want, depending on the matching pattern you set.

Arguments

pattern: It takes a character string containing a regular expression. The character or sequence of characters that will be matched against specified elements of the string.

x: The specified string vector.

ignore.case: If **FALSE**, the pattern matching is case sensitive, and if **TRUE**, the case is ignored during matching.

value: If it is a FALSE, a vector containing the (integer) indices of the matches determined by grep() is returned, and if TRUE, a vector containing the matching elements is returned.

Example:

```r
# Creating string vector
x <- c("CAR", "car", "Bike", "BIKE")

# Calling grep() function
grep("car", x)
grep("Bike", x)
grep("car", x, ignore.case = FALSE)

# to return the vector indices of  both matching elements
grep("Bike", x, ignore.case = TRUE)


# to return matching elements
grep("car", x, ignore.case = TRUE, value = TRUE)
```

Output

```
[1] 2
[1] 3
[1] 2
[1] 3 4
[1] "CAR" "car"
```

Example 2:

```
dollar <- c("I paid $15 for this book.", "they
received
$50,000 in grant money", "two dollars")

grep("$", dollar)
```

Output

1 2 3

Note that it the example above you have three different sentences, two of them use the $ sign and one of them uses the word "dollars". Using only $ to match your pattern will yield to all three sentences being returned

```
dollar <- c("I paid $15 for this book.", "they
received
$50,000 in grant money", "two dollars")

grep("\\$", dollar)
```

Output

1 2

if you add the \\ before the $,then it will return only the sentences having the $ sign.

Example 3:

```
# matches all three vector elements
grep("sh", c("ashes", "shark", "bash"), value=TRUE)

#matches only "shark", starting with sh
grep("^sh", c("ashes", "shark", "bash"), value=TRUE)

#matches only "bash", ending with sh
grep("sh$", c("ashes", "shark", "bash"), value=TRUE)
```

Output

ashes shark

bash shark bash

3) gsub() Function

The `gsub()` function in R is used for replacement operations. The function takes the input and substitutes it against the specified values.

gsub(pattern, replacement, x, ignore.case= FALSE/TRUE)

Example 1:

```
#define string
x <- "This is a fun sentence"

#replace 'fun' with 'great'
x <- gsub('fun', 'great', x, ignore.case=TRUE)

#OUTPUT

[1] "This is a great sentence"
```

Example 2:

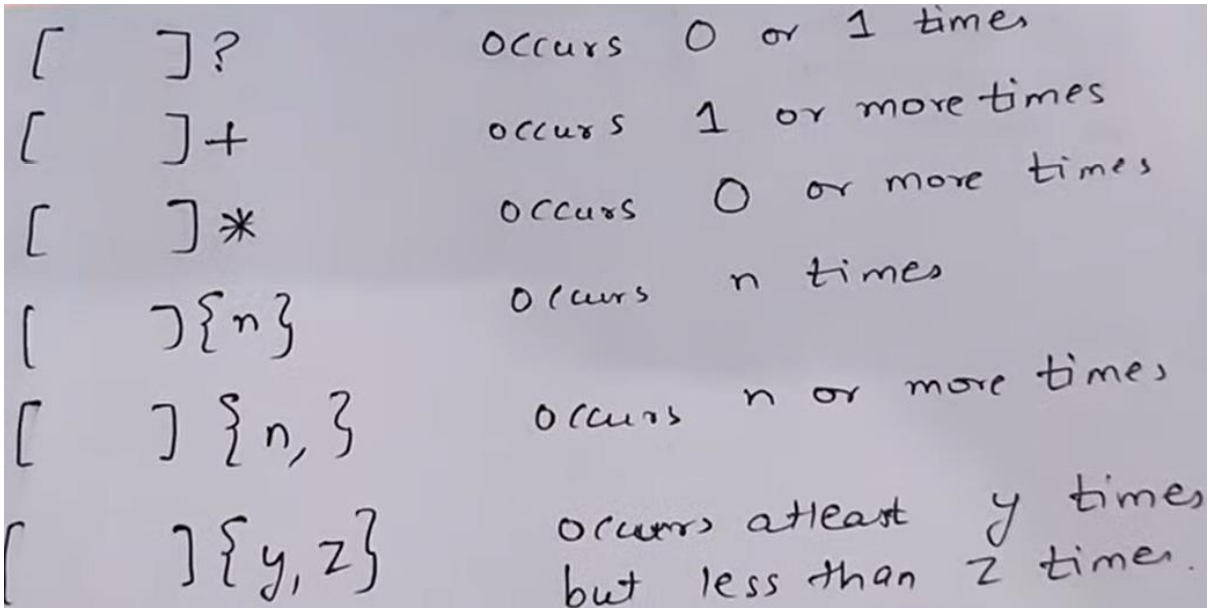
```
ex.sentence <- "Act 3, scene 1. To be, or not to be, that is the
Question:"
```

```
gsub("\w", "*", ex.sentence, ignore.case=TRUE)

Output
"*** *, ***** *. ** **, ** *** ** **, **** ** *** ********:"

gsub("\W", "*", ex.sentence, ignore.case=TRUE)

Output
"Act*3**scene*1**To*be**or*not*to*be**that*is*the*Ques t ion*"
```



FINITE STATE AUTOMATA AUTOMATA

Automaton may be defined as an abstract self-propelled computing device that follows a predetermined sequence of operations automatically.

An automaton having a finite number of states is called a Finite Automaton (FA) or Finite State automata (FSA) .

- Finite automata are used to recognize patterns.
- It takes the string of symbol as input and changes its state accordingly. When the desired symbol is found, then the transition occurs.
- At the time of transition, the automata can either move to the next state or stay in the same state.

- Finite automata have two states, **Accept state** or **Reject state**. When the input string is processed successfully, and the automata reached its final state, then it will accept.

Any regular expression can be implemented as FSA and Any FSA can be described with a regular expression

FINITE AUTOMATON (FA) or FINITE STATE AUTOMATON (FSA)

Mathematical Definition of Automaton

Mathematically, an automaton can be represented by a 5-tuple (Q, Σ, δ, q0, F), where −

Q is a finite set of states.

Σ is a finite set of symbols, called the alphabet of the automaton.

δ is the transition function

q0 is the initial state from where any input is processed (q0 ∈ Q).

F is a set of final state/states of Q (F ⊆ Q).

TYPES OF FINITE STATE AUTOMATON (FSA)

Finite state automation is of two types. Let us see what the types are.

1 DETERMINISTIC FINITE AUTOMATON (DFA)

DFA is defined as the type of finite automation wherein, for every input symbol we can determine the state to which the machine will move. It has a finite number of states that is why the machine is called Deterministic Finite Automaton (DFA).

Mathematically, a DFA can be represented by a 5-tuple (Q, Σ, δ, q0, F), where −

Q is a finite set of states.

Σ is a finite set of symbols, called the alphabet of the automaton.

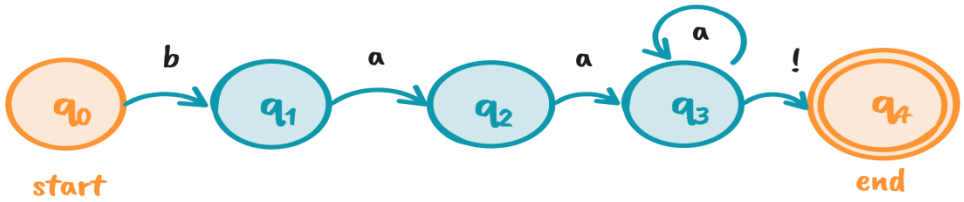
δ is the transition function where δ: Q × Σ → Q .

q0 is the initial state from where any input is processed (q0 ∈ Q).

F is a set of final state/states of Q (F ⊆ Q).

Example:

```
baa+!
```



FSA can be defined as:

$$A = (Q, \Sigma, q_0, F, \delta(q, i))$$

where:

- Q: is a finite set of N states $(\{q_0, q_1, q_2, q_3, q_4\})$
- Σ: finite input alphabet of symbols $(\{a, b, !\})$
- q_0: the designated start state (q_0)
- F: the set of final states, $F \subseteq Q$ $(F = \{q_4\})$
- $\delta(q, i)$: the transition function, *will be discussed in the next section*

2 NON-DETERMINISTIC FINITE AUTOMATON (NDFA)

It may be defined as the type of finite automation where for every input symbol we cannot determine the state to which the machine will move i.e. the machine can move to any combination of the states. It has a finite number of states that is why the machine is called Non-deterministic Finite Automation (NDFA).

Mathematically, NDFA can be represented by a 5-tuple (Q, Σ, δ, q0, F), where − Q is a finite set of states.

Σ is a finite set of symbols, called the alphabet of the automaton.

δ :-is the transition function where δ: Q × Σ → 2 Q.

q0 :-is the initial state from where any input is processed (q0 ∈ Q).

F :-is a set of final state/states of Q (F ⊆ Q).

3 NON-DETERMINISTIC FINITE AUTOMATON with Epsilon

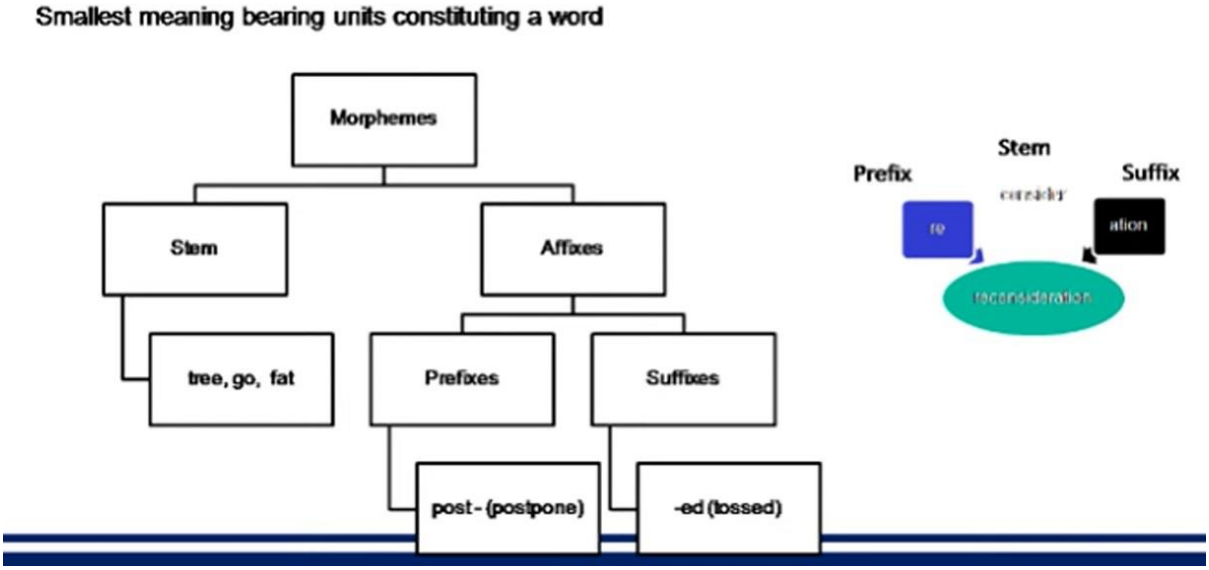
∈-NFA is defined in 5 tuple representation {Q, q0, Σ, δ, F} where
Q is the set of all states,
q0 is the initial state, Σ is
the set of input symbols,

δ is the transition function which is δ:Q × (ΣU∈)->2Q and

F is the set of final states.

Morphology

Morphology is the study of the internal structure of words. Morphology focuses on how the components within a word (stems, root words, prefixes, suffixes, etc.) are arranged or modified to create different meanings.



Types of morphology:

The following are the broad classes of morphology;

- Inflectional morphology - Inflection creates different forms of the same word
- Derivational morphology - Derivation creates different words from the same lemma
- Compounding - Combination of multiple word stems together
- Cliticization - Combination of a word stem with a clitic

Inflection Morphology

Inflection (stem + grammar affix) results in a word of the same class, word does not change its grammatical class

Eg: (walk → walking)

Inflectional Morphology

Word stem + grammatical morpheme

Morpheme serves a syntactic function: grammatical agreement, case marker

- -s for plural on nouns, -ed for past tense on verbs

Results in word of the same class as the stem

- (bat → bats; man → man's; jump → jumps, jumped, jumping)

Noun morphology:

- Regular nouns: bat → bats, fish → fishes
- Irregular nouns: spelling changes (mouse → mice; ox → oxen)

Verb morphology:

- Regular verbs: jump → jumps (-s form), jumped (past/ -ed participle), jumping (-ing participle)
- Irregular verbs: eat → eats (-s form), ate (past), eaten (-ed participle), eating (-ing participle)

Derivation Morphology

Derivation (stem + grammar affix) results in a different class of word; word changes its grammatical form.

Eg: -ation suffix makes verbs into nouns (eg. computerize → computerization)

-ly suffix makes adjectives into adverbs(eg. beautiful-> beautifully)

Derivational Morphology

Word stem + grammatical morpheme

- Usually produces word of **different** class
- More complicated than inflectional

E.g. verbs --> nouns

- -ize verbs → -ation nouns
- generalize, realize → generalization, realization

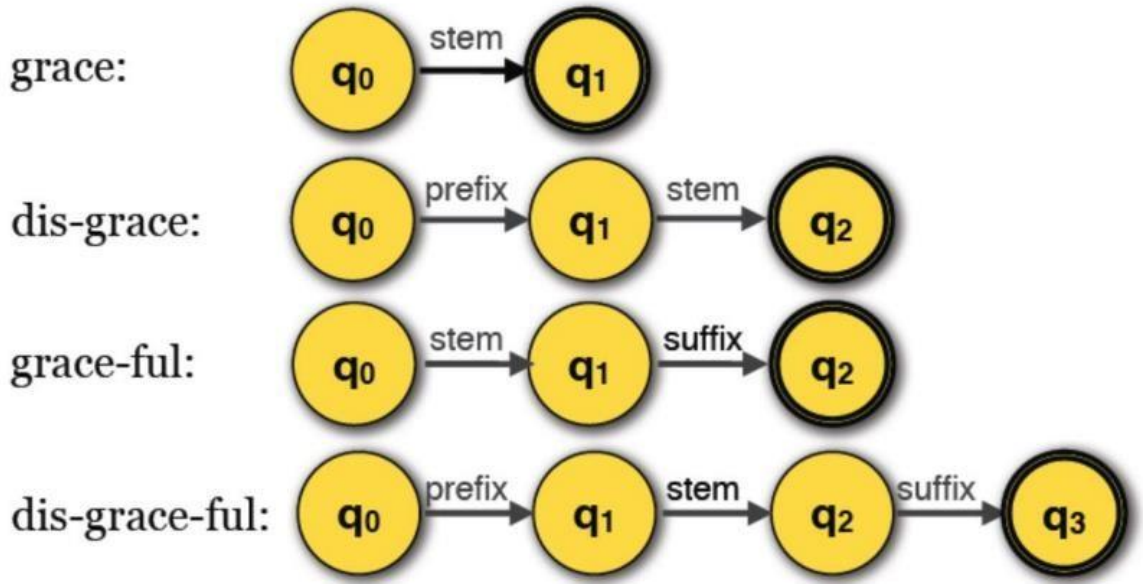
E.g.: verbs, nouns → adjectives

- embrace, pity→ embraceable, pitiable
- care, wit → careless, witless

E.g.: adjective → adverb

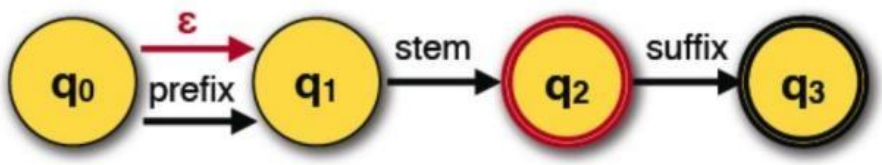
- happy → happily

Finite state automata for morphology



The above finite automata can be merged into the following:

grace,
dis-grace,
grace-ful,
dis-grace-ful



Finite-State Morphological Parsing

Components of Morphological Parsing:

1. **Lexicon**: the list of stems and affixes, together with basic information about them (Noun stem or Verb stem, etc.)

Eg: Cat=Cat + N + SL

2. **Morphotactics**: the model of morpheme ordering that explains which classes of morphemes can follow other classes of morphemes inside a word. E.g., the rule that English plural morpheme follows the noun rather than preceding it.

Eg: Cat-Cats

3. **Orthographic rules**: these spelling rules are used to model the changes that occur in a word, usually when two morphemes combine

(e.g., the y→ie spelling rule changes city + -s to cities).

Lexicon

Stores basic information about a word

Word is stem or a affix?

If stem, then whether a Verb Stem or Noun Stem

If affix, then whether a prefix, infix or suffix

Morphotactic

Set of Rules to make decisions

Decides a word appear/not appear before, after or in between other words

For eg.

use able ness → useableness — Valid obey rules

able use ness → ableuseness — invalid disobey rules

Orthographic rules

Set of rules used to decide spelling changes

For eg.

lady + s ➡ ladys

lady + s ➡ ladies

knife + s ➡ knives

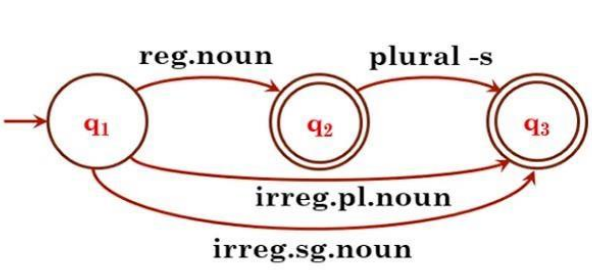
Orthographic Rules

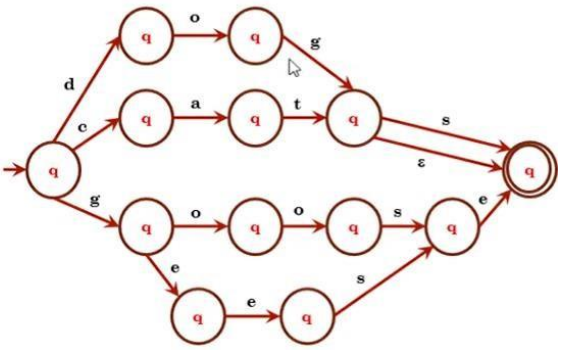
- Consonant Doubling
  - Beg => Begging
- E deletion
  - Make => Making
- E insertion
  - Watch => Watches
- Y replacement
  - Cry => Cried
- K insertion
  - Panic => Panicked

Stem Changes

- Some irregular word requires stem changes
- For eg.
  - goose → geese
  - mouse → mice
  - teach → taught
  - go → went

reg.noun	irreg.pl.noun	irreg.sg.noun	plural
fox	geese	goose	-s
cat	sheep	sheep	
dog	mice	mouse	





We define an intermediate representation which captures morpheme boundaries (^) and word boundaries (#):

Lexicon:	***cat+N+PL***	***fox+N+PL***
⇒ *Intermediate representation:*	***cat^s#***	***fox^s#***
⇒ *Surface string:*	***cats***	***foxes***

Finite-state transducers

An FST $T = L_{in} \times L_{out}$ defines a relation between two regular languages L_{in} and L_{out}:

L_{in} = **{cat, cats, fox, foxes, ...}**

L_{out} = {*cat+N+sg, cat+N+pl, fox+N+sg, fox+N+PL ...*}

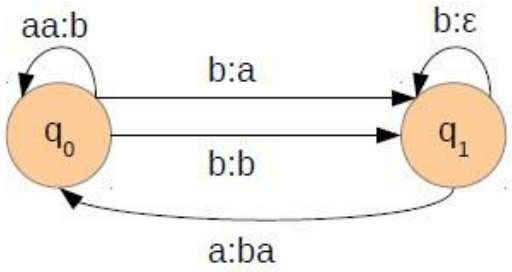
T = { <**cat**, *cat+N+sg*>,
<**cats**, *cat+N+pl*>,
<**fox**, *fox+N+sg*>,
<**foxes**, *fox+N+pl*> }

A **finite-state transducer** $T = \langle Q, \Sigma, \Delta, q_0, F, \delta, \sigma \rangle$ consists of:

- A finite **set of states** $Q = \{q_0, q_1, .., q_n\}$
- A finite alphabet Σ of **input symbols** (e.g. $\Sigma = \{a, b, c, ...\}$)
- A finite alphabet Δ of **output symbols** (e.g. $\Delta = \{+N, +pl, ...\}$)
- A designated **start state** $q_0 \in Q$
- A set of **final states** $F \subseteq Q$
- A **transition function** $\delta: Q \times \Sigma \rightarrow 2^Q$
  $\delta(q,w) = Q'$ for $q \in Q$, $Q' \subseteq Q$, $w \in \Sigma$
- **An output function** $\sigma: Q \times \Sigma \rightarrow \Delta^*$
  $\sigma(q,w) = \omega$ for $q \in Q$, $w \in \Sigma$, $\omega \in \Delta^*$
  If the current state is q and the current input is w, write ω.

Finite-state transducers (FST)

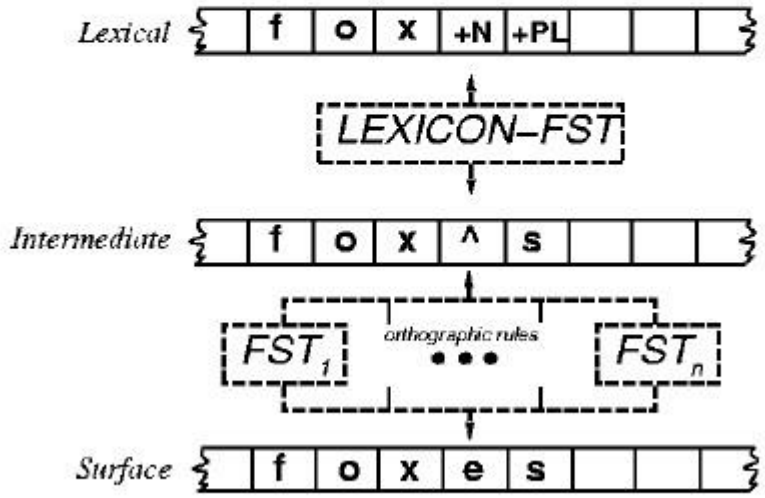
- FST is a type of FSA which maps between two sets of symbols.
- It is a two-tape automaton that recognizes or generates pairs of strings, one from each type.
- FST defines relations between sets of strings.



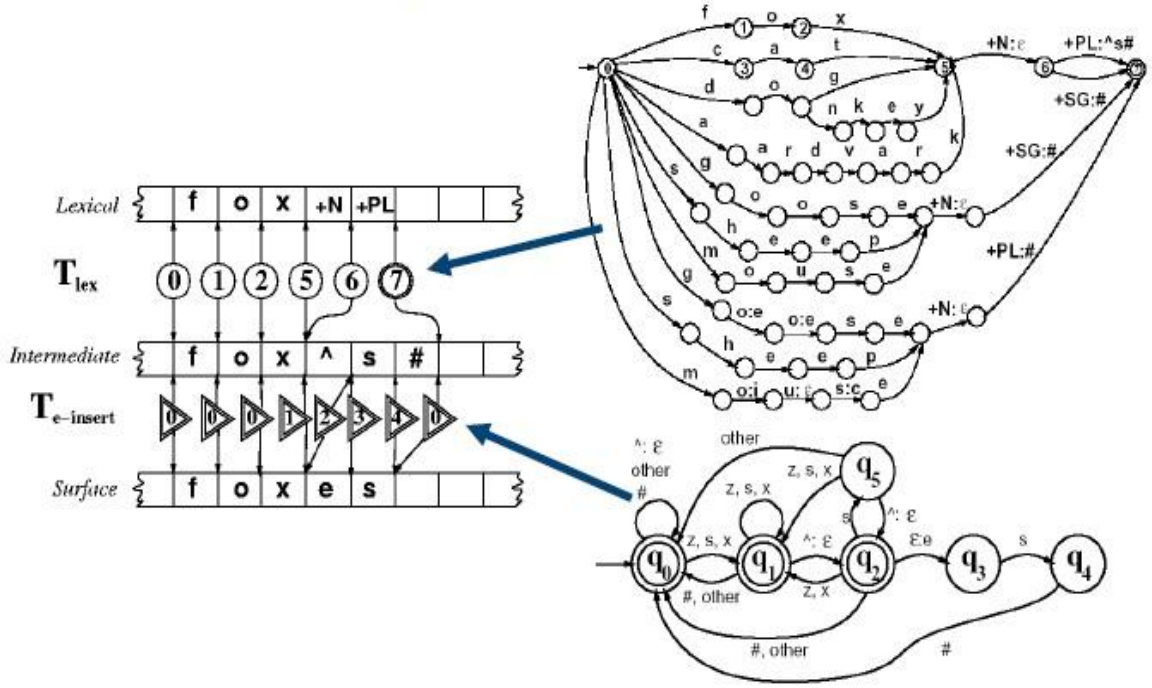
FST for morphological parsing

- Two tapes
  - Upper (lexical) tape: input alphabet Σ
    - cat +N +Pl
  - Lower (surface) tape: output alphabet Δ
    - cats

Combining FST Lexicon and Rules



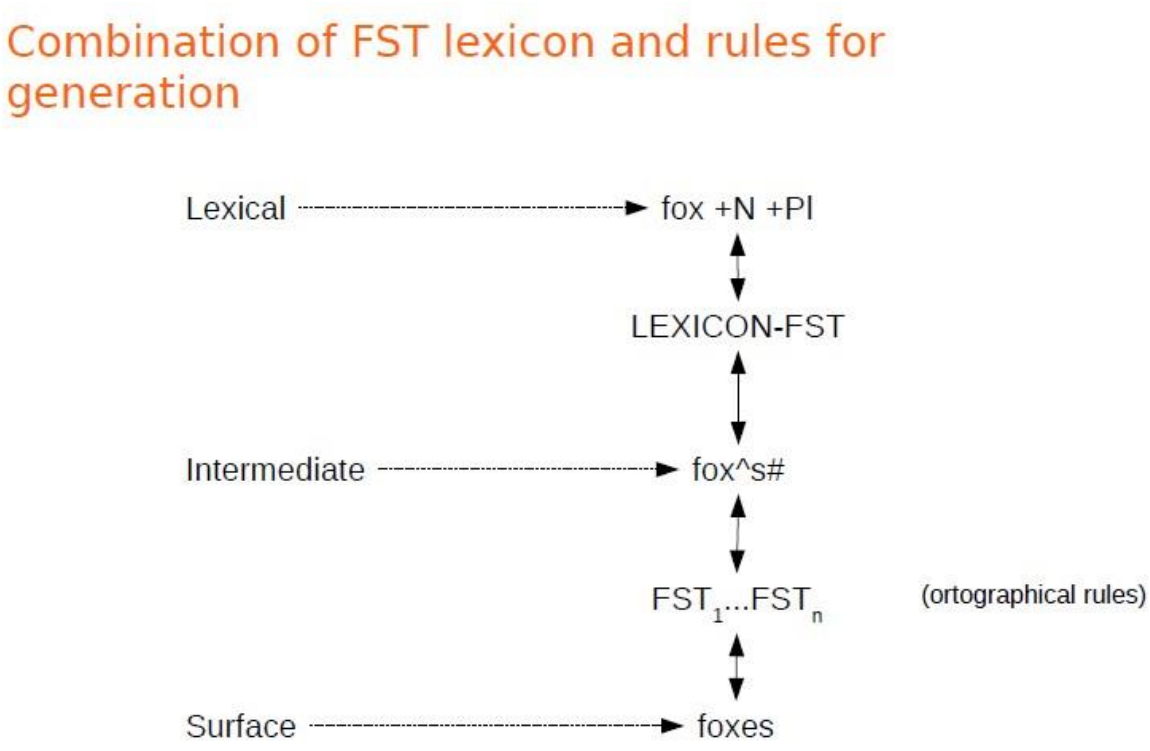
Combining FST Lexicon and Rules



FST and ortographical rules

- Plural of „fox“ is „foxes“ not „foxs“

- Consonant double: beg/begging
- E deletion: make/making
- E insertion: watch/watches
- Y replacement: try/tries
- K insertion: panic/panicked



Stemming

Stemming is an important part of the pipelining process in Natural language processing. Stemming is a natural language processing technique that is used to reduce words to their base form, also known as the root form. The process of stemming is used to normalize text and make it easier to process. It is an important step in text pre-processing, and it is commonly used in information retrieval and text mining applications.

There are several different algorithms for stemming, including the Porter stemmer, Snowball stemmer, and the Lancaster stemmer. The Porter stemmer is the most widely used algorithm, and it is based on a set of heuristics that are used to remove common suffixes from words. The Snowball stemmer is a more advanced algorithm that is based on the Porter stemmer, but it also supports several other languages in addition to English. The Lancaster stemmer is a more aggressive stemmer and it is less accurate than the Porter stemmer and Snowball stemmer.

Suffix Stripping

- It is an operation by which suffixes are removed from terms.

- A document is represented by a vector of words, or terms.
- Terms with a common stem:
  - Connect
  - Connected
  - Connecting
  - Connection
  - Connections

Connect
(Root word)

The Porter Stemmer: definitions

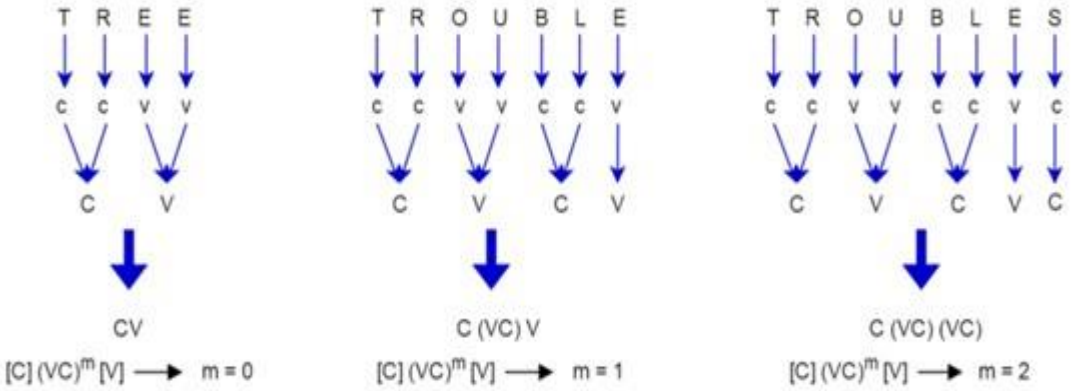
- Definitions:
  - CONSONANT: a letter other than A, E, I, O, U, and Y preceded by consonant
  - VOWEL: any other letter

- With this definition, all words are of the form:
  $(C)(VC)^m(V)$
  C=string of one or more consonants (con+)
  V=string of one or more vowels
  m= measure of word or word part, when represented in this form (VC).

A **consonant** is a letter other than the vowels and other than a letter "Y" preceded by a consonant. So in "TOY" the consonants are "T" and "Y", and in "SYZYGY" they are "S", "Z" and "G".
If a letter is not a consonant it is a **vowel**.
A consonant will be denoted by **c** and a vowel by **v**.

Measure of Word

- M=0 TREE, BY, TR
- M=1 TROUBLE, OATS, TREES, IVY
- M=2 TROUBLES, PRIVATE, OATEN, ORRERY



Rule for removing a suffix

- The rules are of the form:

(condition) S1 -> S2

Where S1 and S2 are suffixes.

This means that if a word ends with a suffix S1, and the stem before S1 satisfies the given condition then S1 is replaced by S2.

- E.g. if rule is (m>1) EMENT ->
  - In this S1 is EMENT and S2 is NULL
  - So, this would map REPLACEMENT to REPLAC

Rule for removing a suffix

- The condition part may also contain the following:

m	The measure of the stem
*S	The stem ends with S
v	The stem contains a vowel
*d	The stem ends with a double consonant (TT, SS)
*o	The stem ends in CVC (second C not W, X, or Y) E.g. WIL, HOP

- The condition part may also contain expressions with and, or and not.
  - E.g. (m>1 and (*s or *t)) : tests for a stem with m>1 ending in s or t

Steps of Port Stemmer Algorithm

The porter stemming algorithm is a process for removing suffixes from words in English.

The Porter Stemmer: Step 1a

- SSES -> SS
  - *caresses -> caress*
- IES -> I
  - *ponies -> poni*
  - *ties -> ti*
- SS -> SS
  - *caress -> caress*
- S -> ϵ
  - *cats -> cat*

The Porter Stemmer: Step 1b

- (m>1) EED -> EE
  - Condition verified: *agreed -> agree*
  - Condition not verified: *feed -> feed*
- (*V*) ED -> ϵ
  - Condition verified: *plastered -> plaster*
  - Condition not verified: *bled -> bled*
- (*V*) ING -> ϵ
  - Condition verified: *motoring -> motor*
  - Condition not verified: *sing -> sing*

If the second or third of the rules in Step 1b is successful, the following is performed.

The Porter Stemmer: (cleanup)

- (These rules are ran if second or third rule in 1b apply)
- AT-> ATE
  - *conflat(ed) -> conflate*
- BL -> BLE
  - *Troubl(ing) -> trouble*
- (*d & ! (*L or *S or *Z)) -> single letter
  - Condition verified: *hopp(ing) -> hop, tann(ed) -> tan*
  - Condition not verified: *fall(ing) -> fall*
- (m=1 & *o) -> E
  - Condition verified: *fil(ing) -> file*
  - Condition not verified: *fail -> fail*

The Porter Stemmer: Step 1C

- Step 1c: Y Elimination (*V*) Y -> I
  - Condition verified: *happy -> happi*
  - Condition not verified: *sky -> sky*

The Porter Stemmer: Step 2

- Step 2: Derivational Morphology, I
  - (m>0) ATIONAL -> ATE
    - *Relational -> relate*
  - (m>0) IZATION -> IZE
    - *generalization-> generalize*
  - (m>0) BILITI -> BLE
    - *sensibiliti -> sensible*

1. (m>0) ATIONAL → ATE
2. (m>0) TIONAL → TION
3. (m>0) ENCI → ENCE
4. (m>0) ANCI → ANCE
5. (m>0) IZER → IZE
6. (m>0) ABLI → ABLE
7. (m>0) ALLI → AL
8. (m>0) ENTLI → ENT
9. (m>0) ELI → E
10. (m>0) OUSLI → OUS
11. (m>0) IZATION → IZE
12. (m>0) ATION → ATE
13. (m>0) ATOR → ATE
14. (m>0) ALISM → AL
15. (m>0) IVENESS → IVE
16. (m>0) FULNESS → FUL
17. (m>0) OUSNESS → OUS
18. (m>0) ALITI → AL
19. (m>0) IVITI → IVE
20. (m>0) BILITI → BLE

The Porter Stemmer: Steps 3

- Step 3: Derivational Morphology, II
  - (m>0) ICATE -> IC
    - *triplicate -> triplic*
  - (m>0) FUL -> ϵ
    - *hopeful -> hope*
  - (m>0) NESS -> ϵ
    - *goodness -> good*

1. (m>0) ICATE → IC
2. (m>0) ATIVE → ε
3. (m>0) ALIZE → AL
4. (m>0) ICITI → IC
5. (m>0) ICAL → IC
6. (m>0) FUL → ε
7. (m>0) NESS → ε

The Porter Stemmer: Step 4

- Step 4: Derivational Morphology, III
  - (m>0) ANCE -> ϵ
    - *allowance-> allow*
  - (m>0) ENT -> ϵ
    - *dependent-> depend*
  - (m>0) IVE -> ϵ
    - *effective -> effect*

1. (m>1) AL → ε
2. (m>1) ANCE → ε
3. (m>1) ENCE → ε
4. (m>1) ER → ε
5. (m>1) IC → ε
6. (m>1) ABLE → ε
7. (m>1) IBLE → ε
8. (m>1) ANT → ε
9. (m>1) EMENT → ε

10. (m>1) MENT → ε
11. (m>1) ENT → ε
12. (m>1 and (*S or *T)) ION → ε
13. (m>1) OU → ε
14. (m>1) ISM → ε
15. (m>1) ATE → ε
16. (m>1) ITI → ε
17. (m>1) OUS → ε
18. (m>1) IVE → ε
19. (m>1) IZE → ε

The Porter Stemmer: Step 5 (cleanup)

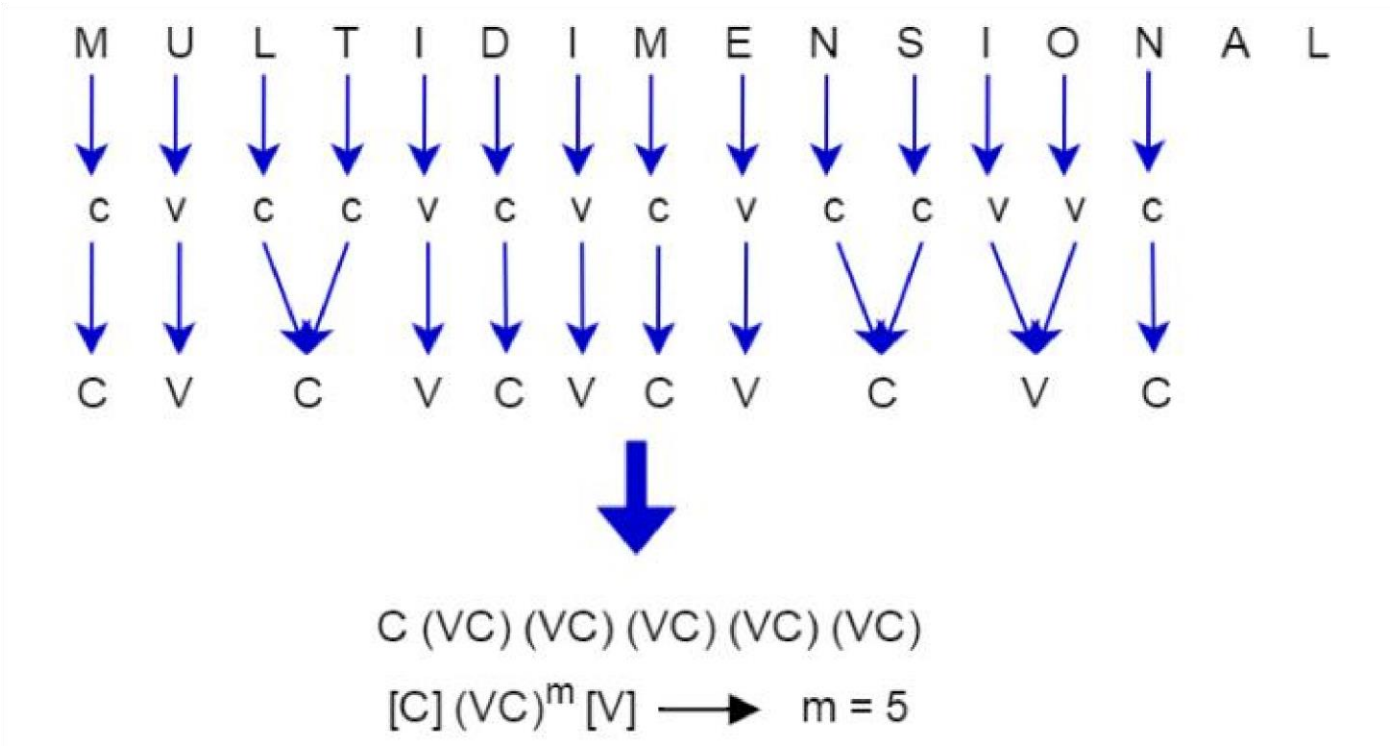
- Step 5a
  - (m>1) E -> ϵ
    - *probate -> probat*
  - (m=1 & !*o) NESS -> ϵ
    - *goodness -> good*
- Step 5b
  - (m>1 & *d & *L) -> single letter
    - Condition verified: *controll -> control*
    - Condition not verified: *roll -> roll*

Example Inputs

Example 1

In the first example, we input the word **MULTIDIMENSIONAL** to the Porter Stemming algorithm. Let's see what happens as the word goes through steps 1 to 5.

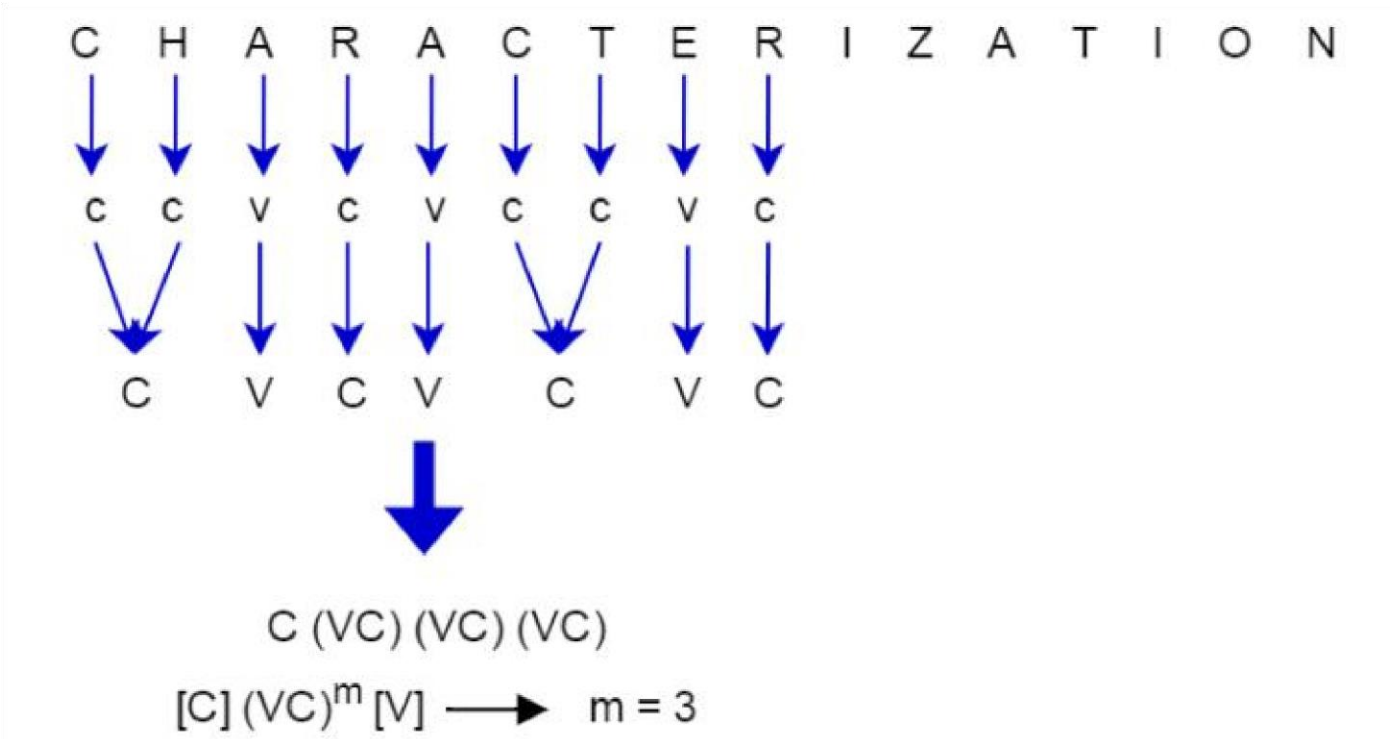
- The suffix will not match any of the cases found in steps 1, 2 and 3. ▪ Then it comes to step 4.
- The stem of the word has m > 1 (since m = 5) and ends with "**AL**".
- Hence in step 4, "**AL**" is deleted (replaced with null).
- Calling step 5 will not change the stem further.
- Finally the output will be **MULTIDIMENSION**. MULTIDIMENSIONAL → **MULTIDIMENSION**



Example 2

In the second example, we input the word **CHARACTERIZATION** to the Porter Stemming algorithm. Let's see what happens as the word goes through steps 1 to 5.

- The suffix will not match any of the cases found in step 1. ▪ So it will move to step 2.
- The stem of the word has m > 0 (since m = 3) and ends with "**IZATION**".
- Hence in step 2, "**IZATION**" will be replaced with "**IZE**". ▪ Then the new stem will be **CHARACTERIZE**.
- Step 3 will not match any of the suffixes and hence will move to step 4.
- Now m > 1 (since m = 3) and the stem ends with **"IZE"**.



- So in step 4, **"IZE"** will be deleted (replaced with null).
- No change will happen to the stem in other steps.
- Finally the output will be **CHARACTER**.
  CHARACTERIZATION → CHARACTERIZE → **CHARACTER**