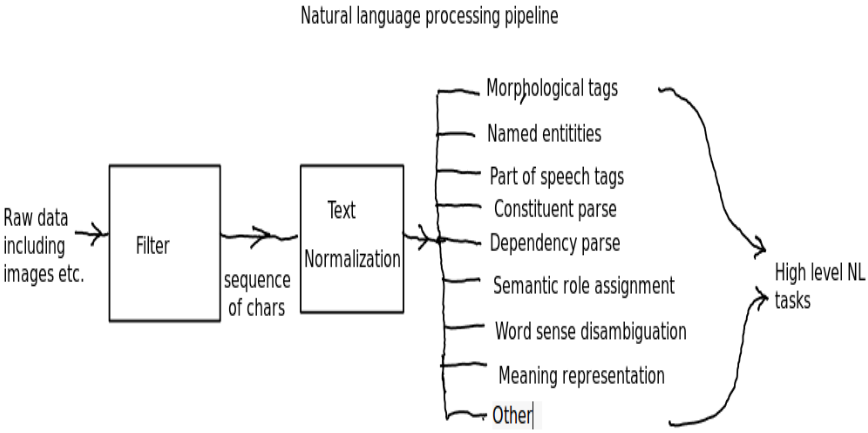
The NL pipeline



Text normalization

- Segmentation
  - Tokenization. Creates tokens - breaks up into words. Can involve merging (New Delhi=single token), split and transform (don't=do not), transform (thru=through)
  - Sentence and paragraph identification.
  - Identification of larger text blocks like sections, chapters etc. (rare).
  - Lemmatization. Break up into root and affixes or other markers. Ex. go, going, went, have the root as 'go', sing, sang, sung, singing have the root 'sing'.
  - Morphome segmentation (needed for morphologically rich languages).
  - Stop word removal.

Regular expressions

- A regular expression is a formal way to specify patterns.
- The expressions are written in a simple language and a driver program or library finds the patterns in text specified by the regular expression.
- It can also be used to replace matching text patterns with other text patterns.
- There are many reg. ex. languages. Each one slightly different from the other. We will look at Python's **re** module. A third party **regex** module in Python provides more complex functionality. Linux has **egrep**, **awk** and **sed**.

Simple reg. expns.

- Characters except the following meta characters:
  `. ^ $ * + ? { } [ ] \ | ( )`
  stand for themselves as patterns.

- `[xyz]` stands for the pattern ‘x’ or ‘y’ or ‘z’. It can also be used to specify ranges. Example: `[b-f]` is the same as `[bcdef]`; similarly `[0-9]` stands for any digit. Most meta characters in lose their meta meaning inside `[...]`. If `^` is the first character within `[]` it complements the set - for example `[^a-f]` stands for all characters except ‘abcdef’. If `^` is not the first character inside `[]` it stands for itself.

- When necessary `\` can be used to escape a character’s meta meaning.

- `\` is also used along with another character to define a set of characters - example: `\w` stands for any alphanumeric character. `\W` matches any non-alphanumeric character. See Python **re** documentation for the full list. They can even be included inside `[]`, so `[\s,.]` will match any white space character (`\s`) or `‘,’` or `‘.’`.

- `‘.’` matches any character except newline.

Grouping and composition

- Concatenation. `This` matches the text 'This'.
- Alternation. `This|That` matches the text 'This' or the text 'That'. Note that concatenation has higher precedence than `|`.
- Grouping. Done by round brackets `( )`. `(Thi(s|T)hat` will match 'Thishat' or 'ThiThat'. Note that it has higher precedence than concatenate or alternate. It is useful when we want to repeat some sub-pattern (next slide).
- `^` matches only at beginning of lines and `$` matches. `^From` will only match 'From' when it occurs at the beginning. So, it will match the 'From' in 'From:hk@iitk.ac.in' but not the 'From' in 'Who is it From?'. `$` works similarly but matches at the end of a line/string.
- `\b` matches only at the begin or end of a word - where a word is defined as a sequence of alphanumeric characters. Example: `\btest\b` will match test in 'the first test was a failure' but not in 'the tests were positive'. Similarly, `\B` matches when current position is not at a word boundary.

Repetition

- `*` stands for repetition 0 or more times. Similarly, + stands for repetition one or more times.
- `?` means 0 or once.
- Repetitions can be more complex - if `R` is a regular expression then the expression `R{m,n}` will match `R` at least m times and at most n times.

Reg. exp. interpreter/library[1]

Method	**Function**
match()	Determine if the RE matches at the beginning of the string
search()	Scan through a string, looking for any location where this RE matches
findall()	Find all substrings where the RE matches, and returns them as a list
finditer()	Find all substrings where the RE matches, and returns them as an iterator

Matching is **greedy**. Matches the longest possible string. These

methods are also available at the top level with more arguments. Example: `re.match(reg.  exp., string)`.

[1] Source: Python online documentation

Match object methods

Method	**Function**
group()	Return the string matched by the ret. exp.
start()	Return the starting position of the match
end()	Return the ending position of the match
span()	Return a tuple containing the (start, end) positions of the match

Grouping,referring back

- Groups formed by ( ) are numbered starting whole reg. exp. as 0 and increasing left to right.
- Matched groups can be referred to using group numbers. Example: \1 refers to the earlier matched pattern string that is group 1. It can be referred to else where in the reg. exp. by \1. Python also allows named groups (using extensions) - see Python documentation.

Modifying

Method	**Function**
split()	Split the string into a list, splitting it wherever the reg. exp. matches
sub()	Find all substrings where the RE matches, and replace them with a different string
subn()	Does the same thing as sub(), but returns the new string and the number of replacements