Natural Language Processing

Lecture 3
Tokenization

2021

Baseline: splitting on whitespace

For many writing systems, splitting the text at *white spaces* is a useful baseline:

'This isn't an easy sentence to tokenize!' $\Rightarrow$
['This', "isn't", 'an', 'easy', 'sentence', 'to', 'tokenize!']

Problems:

- we typically want to treat punctuation marks as separate tokens (but only if they are really punctuation, think of 'U.K.' or '10,000.00$');
- this solution cannot separate token pairs without white space between them, e.g., expressions with clitics like "isn't".

Regular expressions

We need to introduce more sophisticated patterns to describe token boundaries in context-dependent way. A popular solution is using *regular expressions* (regexes for short).

Given a finite Σ alphabet of symbols, regexes over Σ and their matches in Σ^* are defined by simultaneous recursion as follows:

1. The empty string and any single symbol in Σ is a regex over Σ and matches itself.

Regular expressions cont.

2. If r_1 and r_2 are regexes over Σ then
   (a) their *concatenation*, $r_1 r_2$ is also a regex over Σ and matches exactly those strings that are the concatenation of a string matching r_1 and a string matching r_2, and
   (b) their *alternation*, $r_1 | r_2$ is also a regex over Σ and matches exactly those strings that match either r_1 or r_2.
3. If r is a regex over Σ then applying the *Kleene star* operator to r we can form a new regex r^* which matches exactly those strings that are the concatenation of 0 or more strings each matching r.

Regular expressions cont.

A formal language $\mathcal{L}$ (defined simply as an arbitrary set of strings in a finite alphabet) is a *regular language* iff there exists a regular expression that matches exactly $\mathcal{L}$'s elements.

There are simple formal languages that are not regular, e.g., the "twin language" $\{ww \mid w \in \{a, b\}^*\}$.

Nonetheless, regular expressions are flexible enough for a lot of practical tasks, and there are higly efficient algorithms for deciding whether an s string matches a regex (with $\mathcal{O}(\mathrm{length}(s))$ time complexity).

Regular languages and FS acceptors

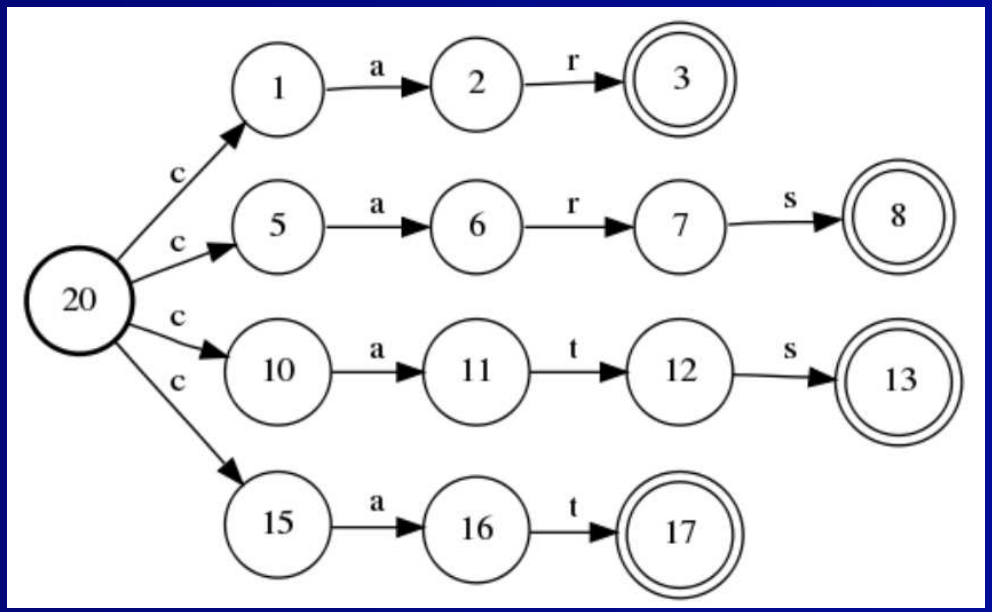
Finite state acceptors are finite state machines that consume an input sequence of characters and can ”accept” or ”reject” the input. They differ from the simplest possible FSA-s by having

- an explicit *start state*,
- a set of designated *accepting states*, and
- their transitions labeled with symbols from a finite alphabet, or with the empty string.

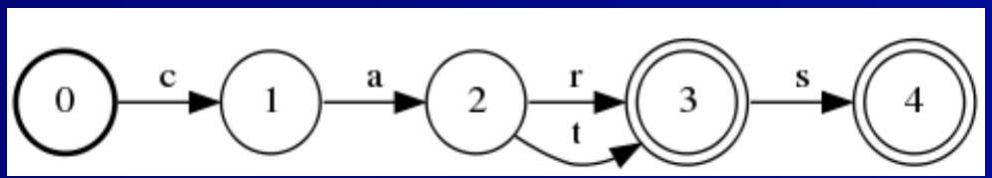
An FS acceptor *accepts* an input iff it has a sequence of transitions which starts from the start state, ends in an accepting state, and the concatenation of the transition labels is the input in question.

Regular languages and FS acceptors cont.

An acceptor for the words ”car”, ”cars”, ”cat” and ”cats”.



Can be simplified to (figures from Butt and Bögel [2009]):



Regular languages and FS acceptors cont.

The connection between FS acceptors and regular languages/expressions is established by Kleene's *Equivalence Theorem*:

- A language is regular iff there is an FSA acceptor that accepts exactly its elements.[1]

This equivalence is important both theoretically and practically: there are very efficient algorithms to simplify/minimize FS acceptors and also to decide whether they accept a string or not.

[1] See, e.g., https://bit.ly/2ZHlKWG for a proof.

Regular expressions: extensions

Convenience extensions not increasing expressive power just adding useful shortcuts, e.g.;

- character classes matching any single symbol in a set, e.g. [0-9];
- complemented character classes matching any character *not* in the complemented set, e.g. [ˆab];
- operator for optional matching: $r?$ matches the empty string or a match of r;
- operators for specifying the required number of pattern repetitions, e.g., $r\{m, n\}$ matches s if s repeats the r pattern k times, with $m \leq k \leq n$.

Regular expressions: back-references

So-called *back-reference* constructions, in contrast, that allow naming and referencing match(es) corresponding to earlier parts of the regex *increase* the expressive power.

For example, most current regex libraries allow a regex similar to

```
(?P<a>[ab]*)(?P=a)
```

which uses back-reference to define exactly the aforementioned non-regular “twin language”.

Regex-based find and replace

In addition to matching whole strings against a regex, two regex-based tasks are very common:

- finding substrings of a string that match a regex,
- regex-based find-and-replace: in its simplest form this is replacing matching substrings with a given string, but two notable extras are provided by modern regex libraries:
  - the regex can have look-ahead and look-back parts, that are used when finding a match but do not count in the part which is replaced;
  - replacements do not have to be fix – they can contain back-references to parts of the match.

Regex cascade-based tokenization

Core idea: perform regex-based substitutions on the input so that in the end it's enough to split on white spaces.

The tokenizer sed script accompanying the Penn Tree Bank is a good example. A few representative rules (\& refers back to the full match, $\backslash n$ to the n-th group):

'...' $\Rightarrow$ ' ... ' (separate ellipsis)
'[,;:#$%&]' $\Rightarrow$ ' \& ' (separate various signs)
\([^.]\)\([.]\)\([])}"']*\)[ ]*$ $\Rightarrow$ '\1 \2\3'
(assume sentence input and split FINAL periods only)
"'ll" $\Rightarrow$ " 'll" (separate clitic 'll)

Regex cascade-based tokenization cont.

The main problem for the approach is the proper handling of exceptions: e.g., word ending periods should be split, *except for abbreviations*.

The standard solution is to replace the problematic expressions with unproblematic placeholders before executing the substitutions in question, e.g.

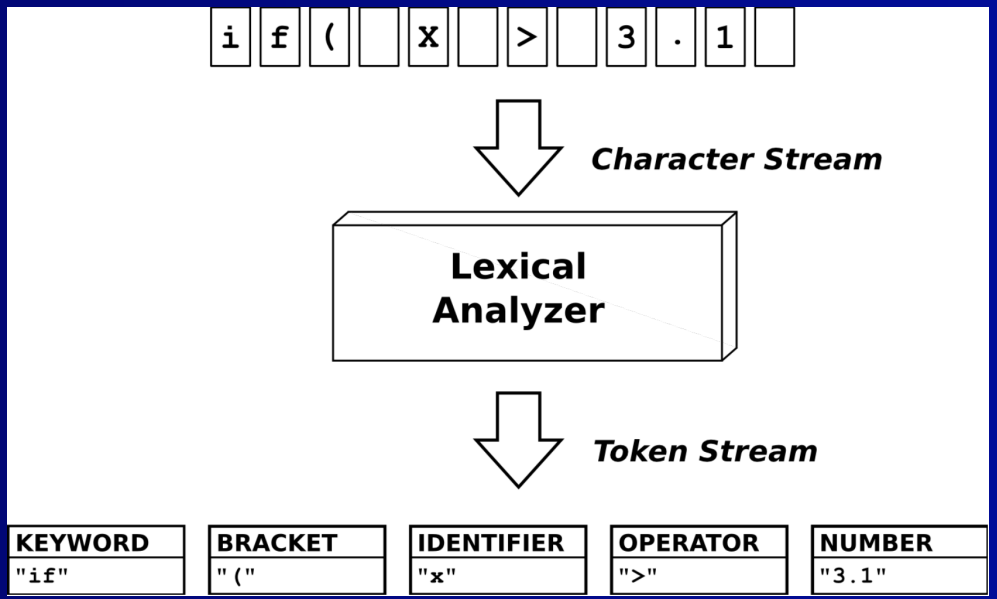
$$(\text{etc}\backslash.|\text{i}\backslash.\text{e}\backslash.|\text{e}\backslash.\text{g}\backslash.) \Rightarrow <\text{abbrev}>$$

This solution requires keeping track of the placeholder substitutions and restoring the originals after executing the problematic rules.

Lexer-based solutions

They use off-the shelf “lexers” (lexical analyzers), originally developed for the tokenization/lexical analysis of computer programs.

A typical lexer takes a character stream as input and produces a stream of classified tokens from it:



Lexer-based solutions cont.

Most lexers are actually lexical analyser generators. Their input is a list of token classes (types), regular expression patterns and

$$[\text{RegexPattern}] \Rightarrow [\text{Action}]$$

rules (where the most important action is classifying the actual match as a token of a given type), and they generate a concrete, optimized lexical analyzer implementing the given rules, e.g., by generating the C source code of the analyzer.

Case study: spaCy's rule-based tokenizer

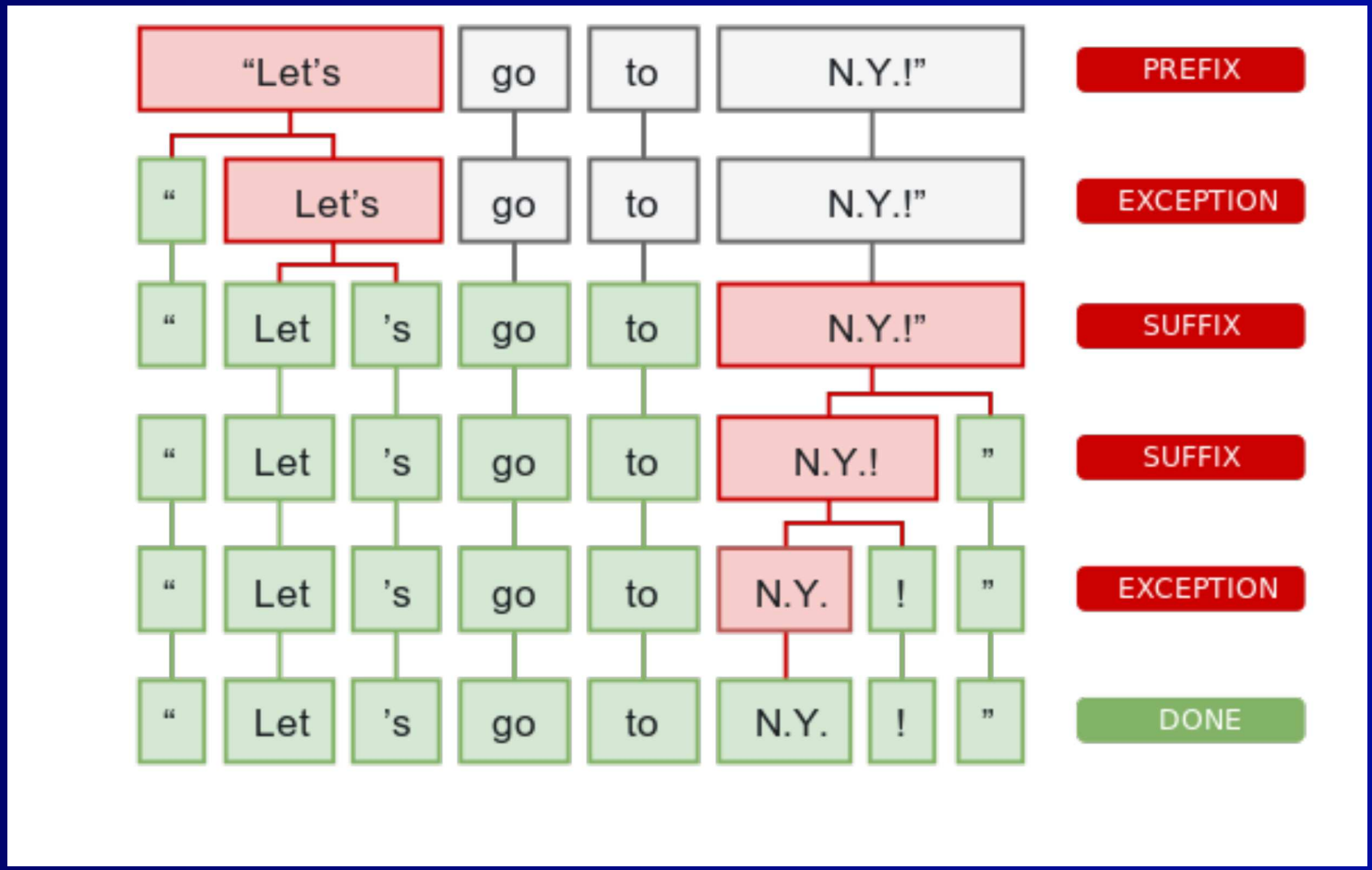
1. The input text is split on white space.[2]
2. Then, the tokenizer processes the text from left to right. On each substring, it performs two checks:
   (a) Does the substring match a tokenizer exception rule? For example, "don't" does not contain whitespace, but should be split.
   (b) Can a prefix, suffix or infix be split off? For example punctuation like commas, periods.

   If there's a match, the rule is applied and the tokenizer continues its loop, starting with the newly split substrings.

[2] The algorithm description is from the spaCy documentation.

SpaCy's rule-based tokenizer cont.

A simple example: tokenizing *"Let's go to N.Y.!"*



(Figure from the spaCy documentation)

Edit distance

In addition to segmenting the input into units, tokenization also involves classifying tokens into types, deciding e.g., which type(s)

'Apple', 'apple', 'appple'

belong to. In many cases, these decisions require a *similarity metric* between strings.
One of the most important metric families in this domain is the so-called *edit distance* family, which measure the distance between two strings by the number edit operations required to transform them into each other.

Edit distance cont.

Given

- a set of *editing operations* (e.g., removing or inserting a character from/into the string) and
- a *weight function* that assigns a weight to each operation,

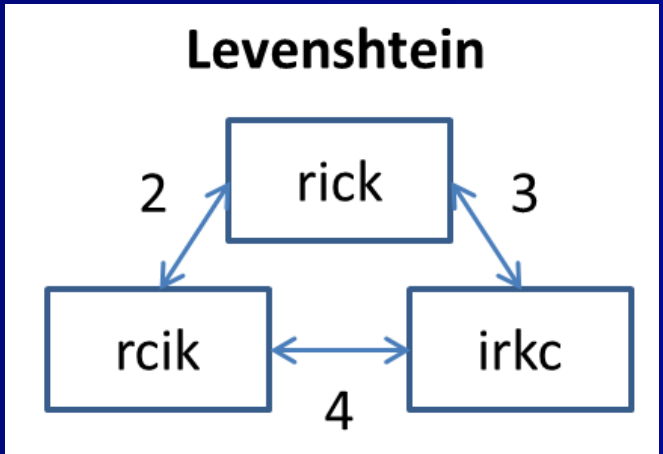
the *edit distance* between two strings, a source and a target, is the minimum total weight that is needed for transforming the source into the target.

Levenshtein distance

One of the most important variants is the so-called Levenshtein distance, where the operations are the

- deletion,
- insertion, and
- substitution of a character,

and the weight of all operations is 1.0.



(Figure from Devopedia)

Tokenization vs subword tokenization

Classical tokenization aims at segmenting the input character stream precisely into linguistically motivated units: words and punctuation. This type of tokenization

- is rather challenging, because there is a wide variety of writing systems and languages that all require (sometimes radically) different rules/models;
- does not necessarily find the most important boundaries, because in many cases *subwords*, i.e. smaller than word units are more useful for downstream tasks;
- generates huge vocabularies on larger corpora, and still leads to problems with out of vocabulary words.

Tokenization vs subword tokenization cont.

A recently developed alternative is *subword tokenization*:

- requires no or only very minimal pretokenization;
- statistical and data-driven: learns the segmentation model from a corpus;
- segments input into frequently occurring word and subword units;
- vocabulary size can be freely chosen, and every token is guaranteed to be in the vocabulary;
- writing and language-agnostic;
- the subword segments are in general informative, boundaries are frequently close to morphological ones.

Tokenization vs subword tokenization cont.

A large number of modern deep, end-to-end NLP architectures use subword-tokenization instead of classical tokenization for segmenting the input.

The main advantages are

- not requiring manual feature engineering or rule writing,
- freely choosable vocabulary size,
- no out-of-vocabulary words,
- providing access to the inner structure of words in many cases: if a word is rare or doesn't occur in the training corpus then it is decomposed into smaller, informative units.

Byte Pair Encoding (BPE)

BPE was originally a simple compression technique for byte sequences, but can be generalized to any sequence consisting of symbols from a finite alphabet. To generate an encoded/compressend version of a sequence over an alphabet,

1. initialize the symbol list with the symbols in the alphabet, and
2. repeatedly count all symbol pairs and replace each occurrence of the most frequent pair (’A’, ’B’) with a new ’AB’ element, and add ’AB’ to the list of symbols.

Byte Pair Encoding (BPE) cont.

How can this technique be used for subword tokenization of texts? The trick is to learn the vocabulary used for segmentation from a training corpus by applying BPE to the text. Modifications to basic BPE:

- start with a rough pretokenization into words (frequently highly simple, e.g. split on white space),
- do not allow BPE merges to cross word boundaries.

In practice these modifications are frequently implemented by adding a new '_' word-end (or word-start) symbol to the alphabet, and stipulating that '_' can only end (or start) merged items.

Byte Pair Encoding (BPE) cont.

A simple example: BPE encoded versions of a sentence after different numbers of merge operations.

1000	to y od a _station is _a _r ail way _station _on _the _ch ū ō _main _l ine
3000	to y od a _station _is _a _railway _station _on _the _ch ū ō _main _line
10000	toy oda _station _is _a _railway _station _on _the _ch ū ō _main _line
50000	toy oda _station _is _a _railway _station _on _the _chū ō _main _line
100000	toy oda _station _is _a _railway _station _on _the _chūō _main _line
Tokenized	toyoda station is a railway station on the chūō main line

(Table from BPEmb: Tokenization-free Pre-trained Subword Embeddings)

As the number of merges increases, more and more symbols are full words.

Greedy BPE subword tokenization

Processing a corpus with BPE results in a 'vocabulary' of all characters together with the results of all merges. New pretokenized input is then subword tokenized by greedily matching the words against this vocabulary/dictionary from left to right:

```
function MaxMatch(string, dictionary) returns list of tokens T

    if string is empty
        return empty list
    for i ← length(sentence) downto 1
        firstword = first i chars of sentence
        remainder = rest of sentence
        if InDictionary(firstword, dictionary)
            return list(firstword, MaxMatch(remainder,dictionary) )
```

(From ch. 2 of Jurafsky and Martin's *Speech and Language Processing* (3rd ed.))

WordPiece

WordPiece is another subword tokenization method that is only slightly different from BPE. The differences are:

- WordPiece works with word-start symbols instead of word-end symbols which is traditional for BPE;
- merges are peformed depending on which resulting merged symbol could be used for a statistical language model with the lowest perplexity (maximal likelihood) on the training corpus. (These concepts will be explained in detail in a later lecture.)

Subword sampling

The default subword tokenization strategy of BPE and WordPiece deterministically produces the greedily matching decomposition of words, even if there are informative alternative segmentations in terms of the vocabulary, e.g.,

$$\textit{unrelated} = \textit{un} + \textit{related}$$

To solve this problem, solutions were developed to *probabilistically sample* from the possible alternative decompositions: *Subword regularization* for WordPiece and *BPE dropout.*[3]

[3] See Kudo: Subword Regularization and Provilkov et al.: BPE-Dropout.

SentencePiece

In their original form, BPE and WordPiece require (crude) pretokenization as a preprocessing step. The SentencePiece tokenizer, in contrast, treats every character, even spaces in the same way, and applies BPE or WordPiece on raw sentences or even paragraphs eliminating the need for pretokenization.
As a result, SentencePiece is one of the most popular solutions for generating the input for deeep end-to-end models from raw text.

References

Miriam Butt and Tina Bögel. Finite state morphology tutorial, 2009. URL https://bit.ly/3sm44fx.