From words to numbers

Parsing, tokenization, extract information

Natural Language Processing

Piotr Fulmański

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
University of Lodz

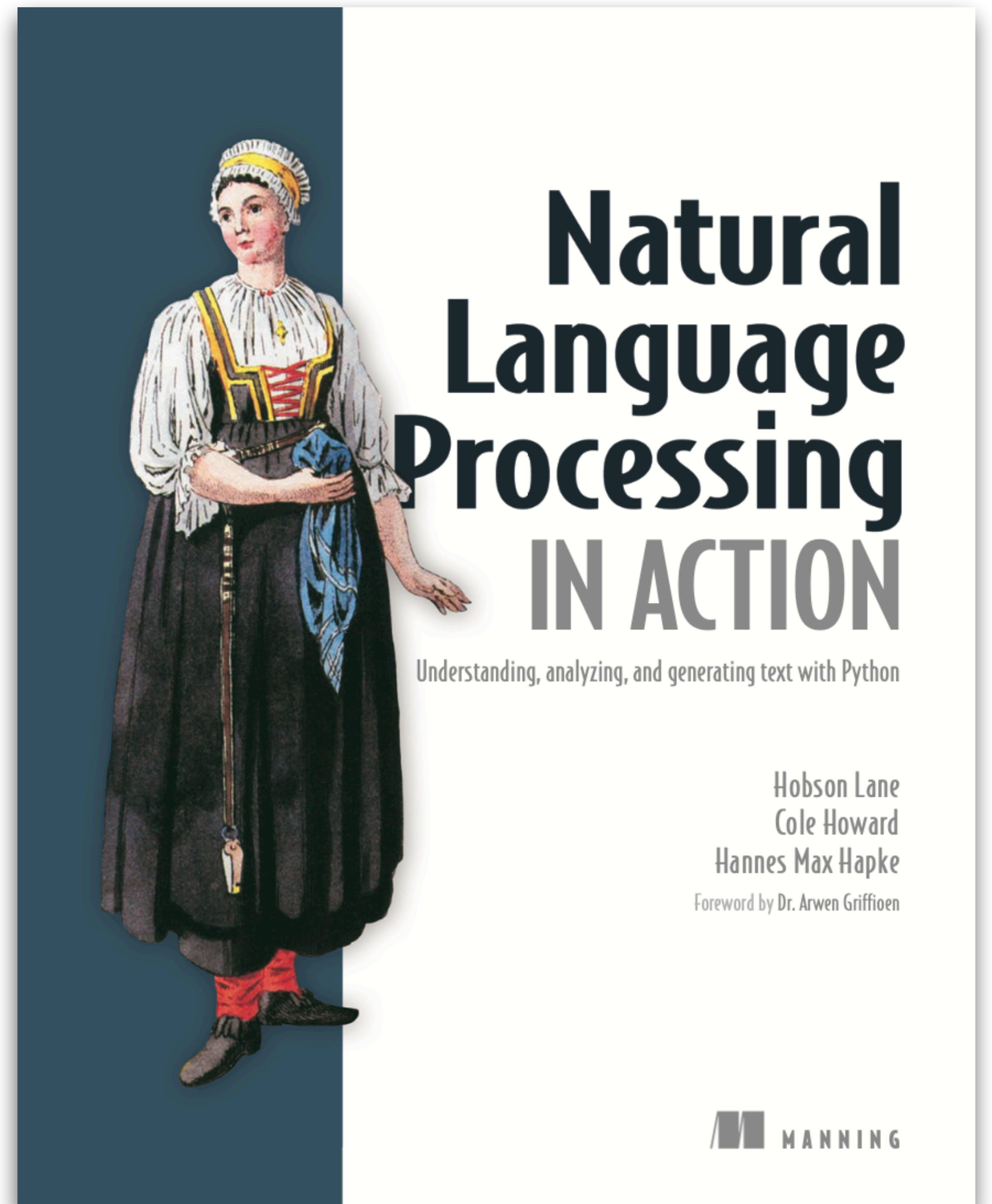
Lecture goals

- Tokenizing your text into words and n-grams (tokens)
- Compressing your token vocabulary with stemming and lemmatization
- Building a vector representation of a statement

Natural Language Processing in Action

by Hobson Lane
Cole Howard
Hannes Max Hapke

Manning Publications, 2019



Terminology

Terminology

- A **phoneme** is a unit of sound that distinguishes one word from another in a particular language.
- In linguistics, a **word** of a spoken language can be defined as the *smallest sequence of phonemes that can be uttered in isolation with objective or practical meaning*.
- The concept of "word" is usually distinguished from that of a *morpheme.*
- Every word is composed of one or more morphemes.
- A **morphem** is the smallest meaningful unit in a language even if it will not stand on its own. A morpheme is not necessarily the same as a word. The main difference between a morpheme and a word is that a morpheme sometimes does not stand alone, but a word, by definition, always stands alone.

Terminology

SEGMENTATION

- Text segmentation is the process of dividing written text into meaningful units, such as **words**, sentences, or topics.
- *Word segmentation* is the problem of dividing a string of written language into its component **words**.

Text segmentation, retrieved 2020-10-20, `https://en.wikipedia.org/wiki/Text_segmentation`

Terminology

SEGMENTATION IS NOT SO EASY

- Trying to resolve the question of what a word is and how to divide up text into words we can face many "exceptions":
  - Is “ice cream” one word or two to you? Don’t both words have entries in your mental dictionary that are separate from the compound word “ice cream”?
  - What about the contraction “don’t”? Should that string of characters be split into one or two “packets of meaning?”
  - The single statement “Don’t!” means “Don’t you do that!” or “You, do not do that!” That’s three hidden packets of meaning for a total of five tokens you’d like your machine to know about.

Terminology

SEGMENTATION IS NOT SO EASY

- In English and many other languages using some form of the Latin alphabet, the **space is a good approximation of a word divider (word delimiter)**, although this concept has limits because of the variability with which languages emically regard collocations and compounds. Many English compound nouns are variably written (for example, ice box = ice-box = icebox; pig sty = pig-sty = pigsty) with a corresponding variation in whether speakers think of them as noun phrases or single nouns.

- However, the equivalent to the word space character is not found in all written scripts, and without it word segmentation is a difficult problem. Languages which do not have a trivial word segmentation process include (to mention just a few examples):

  - Chinese, Japanese, where sentences but not words are delimited,

  - Thai and Lao, where phrases and sentences but not words are delimited,

  - and Vietnamese, where syllables but not words are delimited,

  - see also [Pac].

Text segmentation, retrieved 2020-10-20, `https://en.wikipedia.org/wiki/Text_segmentation`

Terminology

SEGMENTATION IS NOT SO EASY

Going further in processing pipeline, we consider stemming and lemmatization - grouping the various inflections of a word into the same “bucket” or cluster. At first sight simple, it is really difficult thing.

Imagine trying to remove verb endings like “ing” from “ending” so you’d have a stem called “end” to represent both words ("ending" and "end"). And you’d like to stem the word “running” to “run,” so those two words are treated the same. And that’s tricky, because you have to remove not only the “ing” but also the extra “n”. But you want the word “sing” to stay whole. You wouldn’t want to remove the “ing” ending from “sing” or you’d end up with a singleletter “s”.

Or imagine trying to discriminate between a pluralizing “s” at the end of a word like “words” and a normal “s” at the end of words like “bus” and “lens.”

As you can see, even isolated individual letters in a word or parts of a word provide information about that word’s meaning.

Terminology
TOKENIZATION

- In computer science, *lexical analysis*, *lexing* or **tokenization** is the process of **converting a sequence of characters into a sequence of tokens (strings with an assigned and thus identified meaning)**.
- To make it clear: tokenization is the **process of demarcating and possibly classifying sections of a string** of input characters.
- A program that performs lexical analysis may be termed a *lexer*, *tokenizer*, or *scanner*, although scanner is also a term for the first stage of a lexer. A lexer is generally combined with a *parser*, which together analyze the syntax of programming languages, web pages, and so forth.

Lexical analysis, retrieved 2020-10-20, `https://en.wikipedia.org/wiki/Lexical_analysis`

Terminology

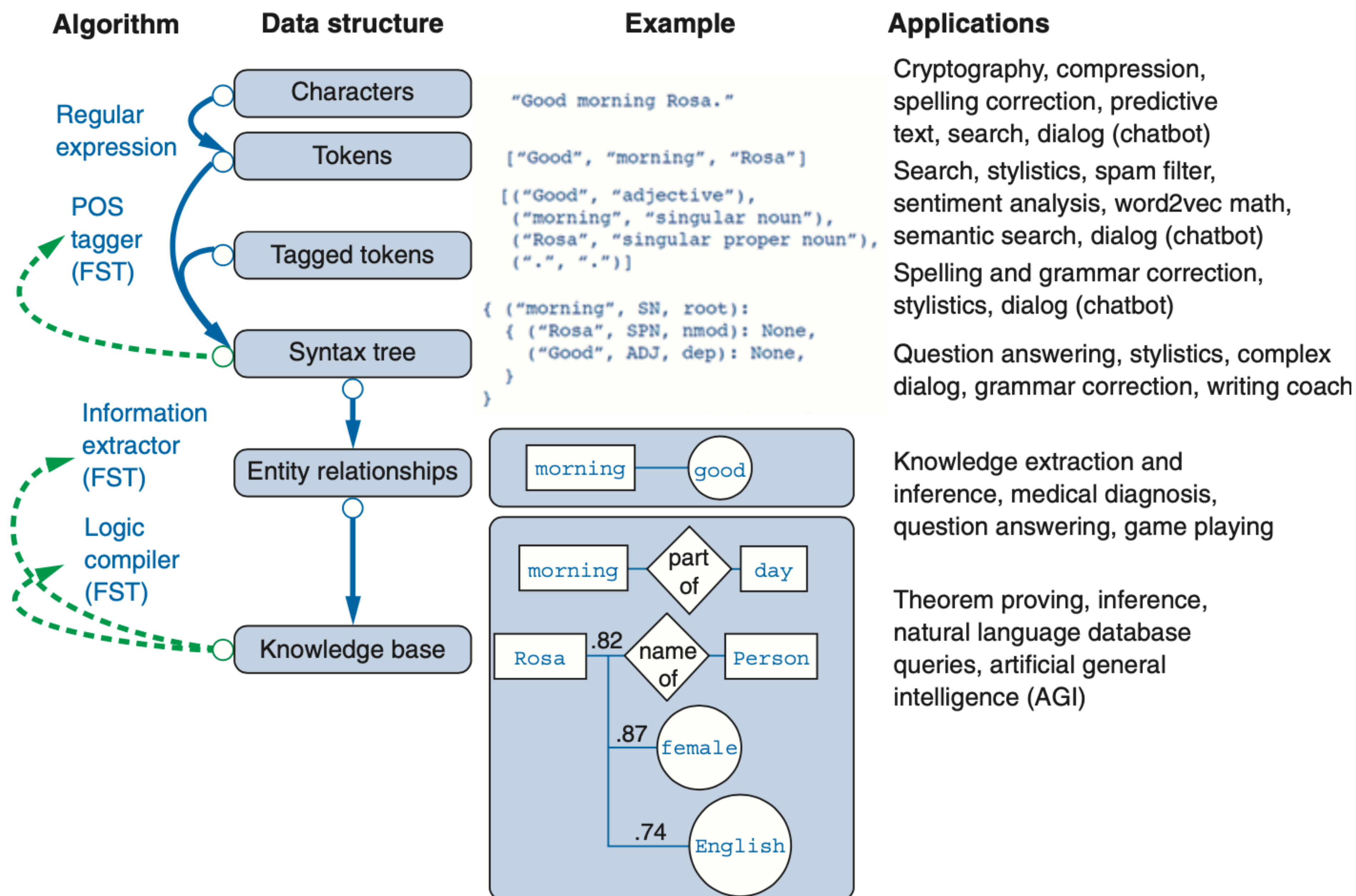
TOKENIZATION

Tokenization is the first step in an NLP pipeline, so it can have a big impact on the rest of your pipeline.

```
tokens -[reg exp?]-> stemming (combine words
with similar meaning) -[assemble]-> vector
representation of your documents
```

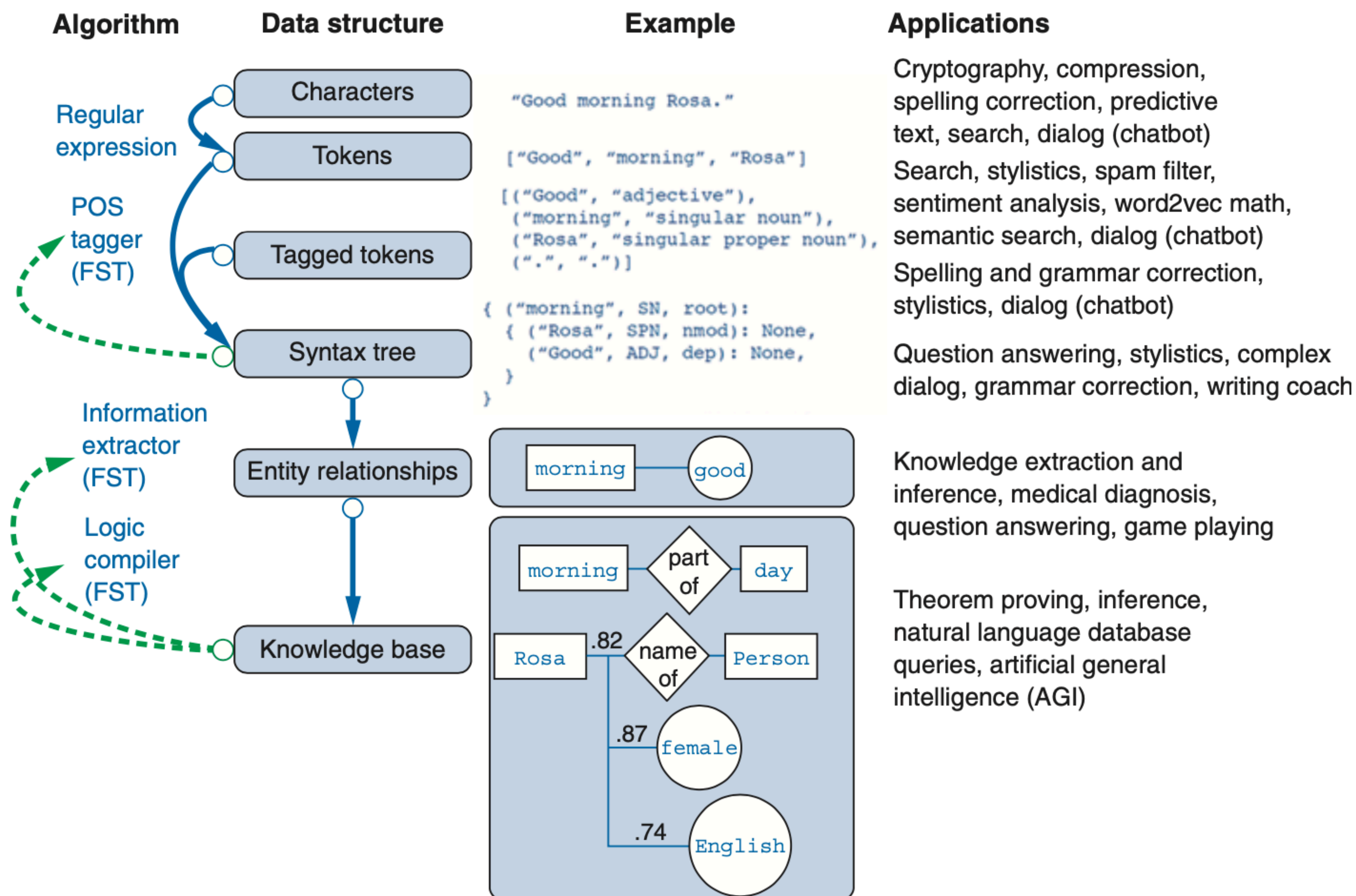
Terminology

NLP PIPELINE



Terminology

NLP PIPELINE



Terminology

TOKENIZATION IS NOT SO EASY

- **Lexers are generally quite simple (really ???)**, with most of the complexity deferred to the *parser* or *semantic analysis* phases, and can often be generated by a lexer generator, notably lex or derivatives.

- **Lexical analysis is also an important early stage in natural language processing**, where text or sound waves are segmented into words and other units. This **requires a variety of decisions which are not fully standardized**, and the number of tokens systems produce varies for strings like "1/2", "chair's", "can't", "and/or", "1/1/2010", "2x4", "...,", and many others. This is in contrast to lexical analysis for programming and similar languages where exact rules are commonly defined and known.

Lexical analysis, retrieved 2020-10-20, `https://en.wikipedia.org/wiki/Lexical_analysis`

Terminology

TOKEN OR LEXEM

- A **lexeme is a sequence of characters that matches the pattern for a token** and is identified by the lexical analyzer as an instance of that token.
- Some authors term this a "token", using "token" interchangeably to represent the string being tokenized, and the *token data structure* resulting from putting this string through the tokenization process.
- The word lexeme in computer science is defined differently than lexeme in linguistics. A lexeme in computer science roughly corresponds to a word in linguistics, although in some cases it may be more similar to a *morpheme*.

Lexical analysis, retrieved 2020-10-20, `https://en.wikipedia.org/wiki/Lexical_analysis`

Terminology
TOKEN

- A token or lexical token is a **string with an assigned and thus identified meaning**. It is structured as a pair consisting of a token name and an optional token value. The token name is a category of lexical unit. Common token names are:
    - identifier: names the programmer chooses;
    - keyword: names already in the programming language;
    - separator (also known as punctuators): punctuation characters and paired-delimiters;
    - operator: symbols that operate on arguments and produce results;
    - literal: numeric, logical, textual, reference literals;
    - comment: line, block.

Lexical analysis, retrieved 2020-10-20, `https://en.wikipedia.org/wiki/Lexical_analysis`

Terminology

TOKEN

Consider this following expression (in the C programming language):

```
x = a + b * 2;
```

The lexical analysis of this expression yields the following sequence of tokens:

```
[(identifier, x), (operator, =), (identifier, a),
(operator, +), (identifier, b), (operator, *),
(literal, 2), (separator, ;)]
```

A token name is what might be termed a part of speech in linguistics.

Lexical analysis, retrieved 2020-10-20, `https://en.wikipedia.org/wiki/Lexical_analysis`

Terminology

PARSER

- *Parsing*, *syntax analysis*, or *syntactic analysis* is the process of **analyzing a string of symbols, conforming to the rules of a formal grammar**. The term parsing comes from Latin *pars*, meaning *part*.
- Parsing, involves breaking down a text into its component parts of speech with an **explanation of the form, function, and syntactic relationship of each part**.
- Within computational linguistics the term is used to refer to the formal analysis by a computer of a sentence or other string of words into its constituents, resulting in a parse tree showing their **syntactic relation to each other**, which may also contain semantic and other information.

Terminology

PARSER

Strictly speaking, a lexer is also a type of parser. The syntax of many programming languages is broken down into two parts:

- **lexical syntax** (the internal construct of the token) which is processed by the **lexer**

- and **grammatical syntax** which is processed by the **parser**.

Lexical syntax is usually a regular expression whose alphabet consists of single characters of the source code.

Grammatical syntax is usually a context-free language whose alphabet consists of tokens produced by a lexer.

More practical part

Tokenization and parsing

Task 1

BUILD A TOKENIZER

- From the ground
- Lex - A Lexical Analyzer Generator
- Yacc - Yet Another Compiler-Compiler
- Flex - a fast scanner generator
- Apache OpenNLP
  OpenNLP supports the most common NLP tasks, such as tokenization, sentence segmentation, part-of-speech tagging, named entity extraction, chunking, parsing, language detection and coreference resolution.
- spaCy - Accurate , flexible, fast, Python
- Stanford CoreNLP - More accurate, less flexible, fast, depends on Java 8
- NLTK - Standard used by many NLP contests and comparisons, popular, Python

Lexical analysis, retrieved 2020-10-20, `https://en.wikipedia.org/wiki/Lexical_analysis`

Task 1

BUILD A TOKENIZER

- See also: [LexYac]

Lexical analysis, retrieved 2020-10-20, `https://en.wikipedia.org/wiki/Lexical_analysis`

Task 1

BUILD A TOKENIZER

- Tokenization is the first step in an NLP pipeline, so it can have a big impact on the rest of your pipeline:

```
tokens --> stemming --> vectors
```

- The simplest way to tokenize a sentence is to use whitespace within a string as the "delimiter" of words.

Lexical analysis, retrieved 2020-10-20, `https://en.wikipedia.org/wiki/Lexical_analysis`

Task 2

BUILD A PARSER

- Having real tokens (string + meaning understood as a role) we can parse a sentence, i.e. check its grammatical syntax (if it's in accordance with some rules).
- We can do this with FSM (Finite State Machine).

Lexical analysis, retrieved 2020-10-20, `https://en.wikipedia.org/wiki/Lexical_analysis`

From tokens to numerical vector representation

One-hot vectors

- In this representation each row of the table is a binary row vector for a single word from sentence. This explains why it’s called a one-hot vector: all but one of the positions (columns) in a row are 0 or blank.
- Order of vectors corresponds to order of words in the sentence.
- This way one-hot vector table is like a recording of the original text.
- One nice feature of this vector representation of words and tabular representation of documents is that no information is lost.
- It's understandable that table (matrix) containing one-hot vectors is super-sparse.
- Apart from all the advantages and disadvantages of such representation the important thing is that you’ve **turned a sentence of natural language words into a sequence of numbers, or vectors**.

One-hot vectors

- But this is a big table for a short sentence. This is almost like a raw “image” of your document. Storing all those zeros, and trying to remember the order of the words in all your documents, doesn’t make much sense. It’s not practical. You **need to do dimension reduction** if you want to extract useful information from the data.
- Fortunately, you don’t ever use this data structure for storing documents.
- You only use it temporarily, in RAM, while you’re processing documents one word at a time. or as input data for some algorithms (for example neural networks).
- And what you really want to do is **compress the meaning of a document down to its essence**.
- You’d like to compress your document down to a **single vector rather than a big table**. You just want to capture most of the meaning (information) in a document, not all of it.

One-hot vectors

CODE

- `lecture_02_01_01.py` - calculate and print one-hot matrix
- `lecture_02_01_02.py` - calculate and print one-hot matrix in more readable way (slightly modified `lecture_02_01_01.py`)

Bag-of-words (BOW)

- To compress the meaning of a document down to its essence we can use bag-of-words vector approach to obtain data structure that's easier to work with.
- We can make an assumption that most of the meaning of a sentence can be gleaned from just the words themselves. Let's assume you can ignore the order and grammar of the words, and jumble them all up together into a "bag".
- If you summed all one-hot vectors together, rather than considering them one at a time, you'd get a **bag-of-words vector**. This is also called a **word frequency vector**, because it only counts the frequency of words, not their order.

Bag-of-words (BOW)

CODE

- `lecture_02_02_01.py` - calculate BOW for one sentence (document).
- `lecture_02_02_02.py` - calculate BOW for multiple sentences (documents).
- `lecture_02_02_02_02.py` - calculate BOW for multiple very simplified sentences (documents); letters are used instead of words (the same code as for `lecture_02_02_02.py`).

Bag-of-words (BOW)

USABILITY OF BAG OF WORDS

Measuring bag-of-words overlap with dotproduct

- `lecture_02_03.py` - calculate similar sentences in terms of the number of common words (number of words used in both sentences).

Extending vocabulary with n-grams

An n-gram is a sequence containing up to *n* elements that have been extracted from a sequence of those elements. In general the "elements" of an n-gram can be characters, syllables, words, etc.

Extending vocabulary with n-grams

Why bother with n-grams? As you saw earlier, when a sequence of tokens is vectorized into a bag-of-words vector, it loses a lot of the meaning inherent in the order of those words.

By extending the concept of a token to include multiword tokens (n-grams), your NLP pipeline can retain much of the meaning inherent in the order of words in your statements.

For example, the meaning-inverting word "not" will remain attached to its neighboring words, where it belongs. Without n-gram tokenization, it would be free floating.

A bit of the context of a word is retained in your pipeline, when you tie it to its neighbor(s).

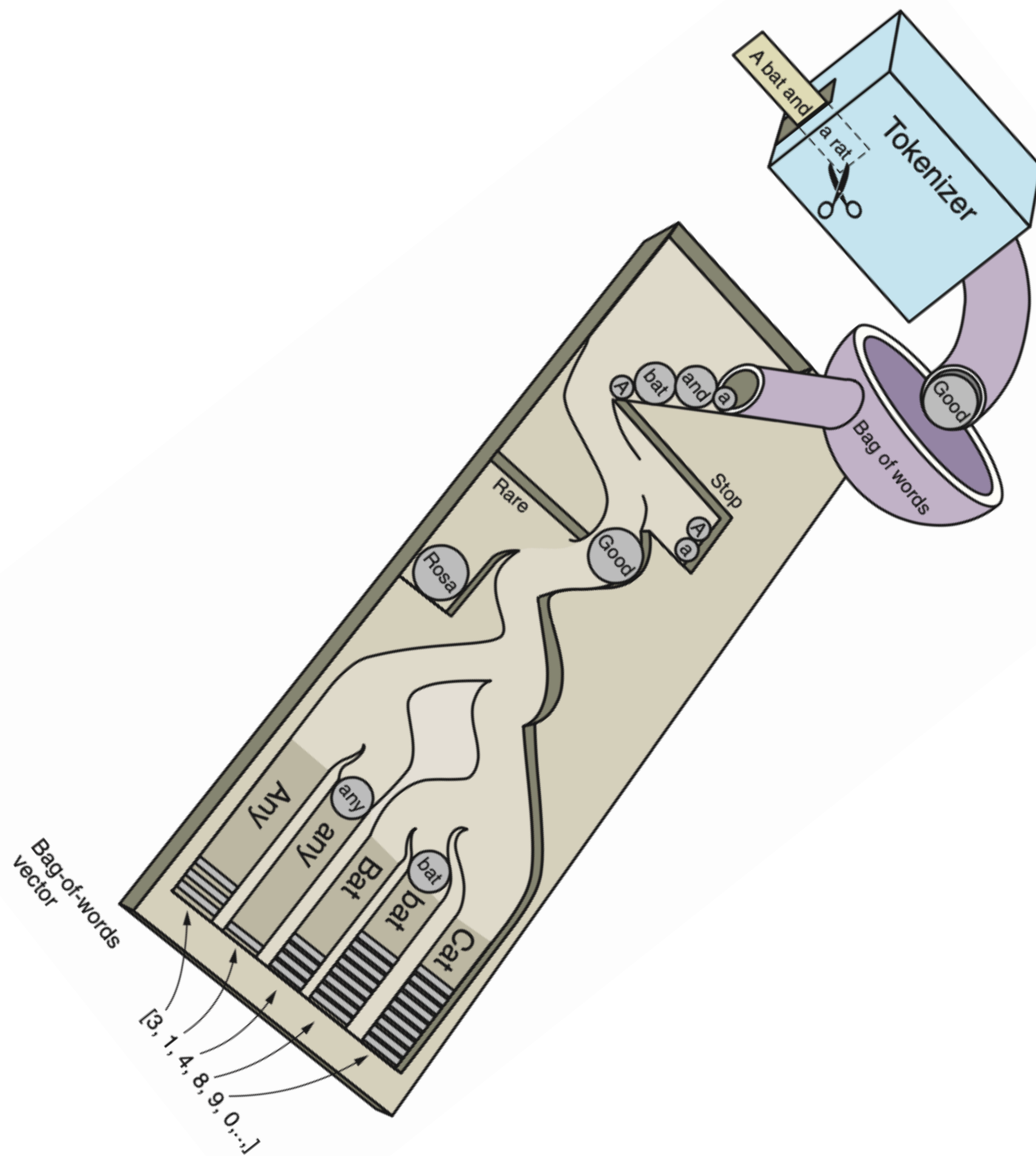
Extending vocabulary with n-grams

- CODE: n-gram code
- `lecture_02_04.py` - generate n-grams

Rare and stop words in BOW

- Rare tokens in most cases are helpful for classification problem. Typically, n-grams are filtered out that occur too infrequently.
- Now consider the opposite problem. Consider the 2-gram “at the”. That’s probably not a rare combination of words. In fact it might be so common, spread among most of documents, that it loses its utility for discriminating between the meanings of your documents. It has little predictive power. Just like words and other tokens, n-grams are usually filtered out if they occur too often.

BOW with rare and stop words



Rare and stop words in BOW

But be careful when remove any part of sentence. Consider these two examples:

- Mark *reported to the CEO*.
- Suzanne *reported as the CEO* to the board.

In your NLP pipeline, you might create 4-grams such as "*reported to the CEO*" and "*reported as the CEO*". If you remove the stop words from the 4-grams, both examples would be reduced to "*reported CEO*", and you would lack the information about the professional hierarchy.

Normalization

Normalization

CASE FOLDING

However, keep in mind that some information is often communicated by capitalization of a word.

Normalization

Stemming and lemmatization

- *Stemming* is the process of reducing inflection in words to their **word stem, base or root forms** such as mapping a group of words to the same stem **even if the stem itself is not a valid word** in the language.
- In computational linguistics, *lemmatisation* is the algorithmic process of determining the *lemma* (a lemma (pl.: *lemmas* or *lemmata*) is the canonical form, dictionary form, or citation form of a set of words) of a word based on its intended meaning.

  Lemmatization, unlike stemming, reduces the inflected words properly ensuring that the **root word belongs to the language**.

Nice description you can find in [SteLem].
Consider also Porter's original stemmer algorithm implemented in pure Python [Por].

Bibliography

- [Lan] Hobson Lane, Cole Howard, Hannes Max Hapke, *Natural Language Processing in Action*, Manning Publications, 2019
- [Pac] J. Packard, I*n The Morphology of Chinese: A Linguistic and Cognitive Approach*, Cambridge University Press, 2000
- [Por] `https://github.com/jedijulia/porter-stemmer/blob/master/stemmer.py`
- [SteLem] Stemming and Lemmatization in Python, retrieved 2020-10-19, `https://www.datacamp.com/community/tutorials/stemming-lemmatization-python`
- [LexYac]
    1. The Lex & Yacc Page,
       retrieved 2020-10-23,
       http://dinosaur.compilertools.net
    2. Writing an Interpreter with Lex, Yacc, and Memphis,
       retrieved 2020-10-23,
       `http://memphis.compilertools.net/interpreter.html`