

# Language Technology

## Chapter 3: Corpus Processing Tools

Pierre Nugues

Pierre.Nugues@cs.lth.se

September 2 and 5, 2024



# Corpora

A corpus is a collection of texts (written or spoken) or speech  
Corpora are balanced from different sources: news, novels, etc.

	English	French	German
<b>Most frequent words in a collection of contemporary running texts</b>	<i>the</i> <i>of</i> <i>to</i> <i>in</i> <i>and</i>	<i>de</i> <i>le</i> (article) <i>la</i> (article) <i>et</i> <i>les</i>	<i>der</i> <i>die</i> <i>und</i> <i>in</i> <i>des</i>
<b>Most frequent words in Genesis</b>	<i>and</i> <i>the</i> <i>of</i> <i>his</i> <i>he</i>	<i>et</i> <i>de</i> <i>la</i> <i>à</i> <i>il</i>	<i>und</i> <i>die</i> <i>der</i> <i>da</i> <i>er</i>



# Characteristics of Current Corpora

Big:

- The Bank of English (Collins and U Birmingham) from the 1990s had more than 500 million words
- Colossal Clean Crawled Corpus (2020) has 156 billion tokens (English)

Available in many languages

Easy to collect: The web is the largest corpus ever built and within the reach of a mouse click

Parallel: same text in two languages: English/French (Canadian Hansards), European parliament (23 languages)

Annotated with part-of-speech or manually parsed (treebanks):

- Characteristics/N of/PREP Current/ADJ Corpora/N
- (NP (NP Characteristics) (PP of (NP Current Corpora)))



# Lexicography

## Writing dictionaries

Dictionaries for language learners should be build on real usage

- *They're just trying to score **brownie points** with politicians*
- *The boss is pleased – that's another **brownie point***

Bank of English: *brownie point* (6 occs) *brownie points* (76 occs)

Extensive use of corpora to:

- Find **concordances** and cite real examples
- Extract **collocations** and describe frequent pairs of words



# Concordances

A word and its context:

Language	Concordances
English	s beginning of miracles did Je n they saw the miracles which n can do these miracles that t ain the second miracle that Je e they saw his miracles which
French	le premier des miracles que fi i dirent: Quel miracle nous mo om, voyant les miracles qu'il peut faire ces miracles que tu s ne voyez des miracles et des



# Collocations

Word preferences: Words that occur together

	English	French	German
<b>You say</b>	<i>Strong tea</i>	<i>Thé fort</i>	<i>Schmales Gesicht</i>
	<i>Powerful computer</i>	<i>Ordinateur puissant</i>	<i>Enge Kleidung</i>
<b>You don't say</b>	<i>Strong computer</i>	<i>Thé puissant</i>	<i>Schmale Kleidung</i>
	<i>Powerful tea</i>	<i>Ordinateur fort</i>	<i>Enges Gesicht</i>



# Word Preferences

Strong w			Powerful w		
<i>strong w</i>	<i>powerful w</i>	<i>w</i>	<i>strong w</i>	<i>powerful w</i>	<i>w</i>
161	0	showing	1	32	than
175	2	support	1	32	figure
106	0	defense	3	31	minority
...					



# Corpora as Knowledge Sources

## Traditional use:

- Describe usage more accurately

## Machine learning

- Learn statistical/machine-learning models for speech recognition, taggers, parsers
- Assess tools: part-of-speech taggers, parsers.
- Derive automatically patterns from annotated or unannotated corpora

## Applications:

- Translation
- Information and knowledge extraction
- Question answering from textual sources





# Finite-State Automata

A flexible tool to search and process text

A FSA accepts and generates strings, here *ac*, *abc*, *abbc*, *abbbc*, *abbbbbbbbbbbbc*, etc.



# FSA

Mathematically defined by

- $Q$  a finite number of states;
- $\Sigma$  a finite set of symbols or characters: the input alphabet;
- $q_0$  a start state,
- $F$  a set of final states  $F \subseteq Q$
- $\delta$  a transition function  $Q \times \Sigma \rightarrow Q$  where  $\delta(q, i)$  returns the state where the automaton moves when it is in state  $q$  and consumes the input symbol  $i$ .



# Regular Expressions

Regexes are equivalent to FSA and generally easier to use

① Constant regular expressions:

Pattern	String
regular	<i>A section on <u>regular</u> expressions</i>
the	<i>The book of <u>the</u> life</i>

② Metacharacters like \*, where the automaton above is described by the regex `ab*c`

```
$ grep 'ab*c' myFile1 myFile2
```

While `grep` was the first regex tool, most programming languages adopt the Perl syntax



# Main Regex Operations

The two main regex operations are match and substitute. They are often abridged using the Perl regex notations where:

- The `m/regex/` construct denotes a match operation with the regular expression `regex`.
- The `s/regex/replacement/` construct is a substitution operation. This statement matches the first occurrence of `regex` and replaces it by the `replacement` string. If we want to replace all the occurrences of a pattern, we use the `g` modifier, where `g` stands for globally: `s/regex/replacement/g`.



# regex101.com

regex101.com: A site to experiment and test regular expressions.

The screenshot displays the regex101.com web application. The interface is divided into several sections:

- REGULAR EXPRESSION:** The input field contains the regex `/ac*e/g`. A green status bar above it indicates "3 MATCHES - 21 STEPS".
- TEST STRING:** The input field contains the text "The **a**erial **ac**celeration alerted the **a**ce pilot". The characters 'a', 'ac', and 'a' are highlighted in blue, indicating matches.
- EXPLANATION:** This section provides details about the matches:
  - `/ac*e/g`:
    - `a` matches the character `a` literally (case sensitive)
    - `c*` matches the character `c` literally (case sensitive)
  - Quantifier:** `*` Between `zero` and
- MATCH INFORMATION:** This section states: "No match groups were extracted." and "This means that your pattern matches but there were no (capturing groups) in it that matched anything in the subject string."
- QUICK REFERENCE:** This section includes links for "FULL REFERENCE" and "MOST USED TOKENS".
- FLAVOR:** A sidebar on the left lists different regex flavors: PCRE (selected), JS, and PY.
- TOOLS:** A sidebar on the left lists various tools: `</>`, `&`, and `✓`.
- SUBSTITUTION:** A field at the bottom for defining replacement strings.

# Metacharacters

Chars	Descriptions	Examples
*	Matches any number of occurrences of the previous character – zero or more	ac*e matches strings ae, ace, acce, accce, etc. as in “The <u>a</u> erial <u>a</u> cceleration alerted the <u>a</u> ce pilot”
?	Matches at most one occurrence of the previous character – zero or one	ac?e matches ae and ace as in “The <u>a</u> erial acceleration alerted the <u>a</u> ce pilot”
+	Matches one or more occurrences of the previous character	ac+e matches ace, acce, accce, etc. as in as in “The aerial <u>a</u> cceleration alerted the <u>a</u> ce pilot”



# Metacharacters

Chars	Descriptions	Examples
<code>{n}</code>	Matches exactly $n$ occurrences of the previous character	<code>ac{2}e</code> matches <code>acce</code> as in “The aerial <u>acceleration</u> alerted the ace pilot”
<code>{n,}</code>	Matches $n$ or more occurrences of the previous character	<code>ac{2,}e</code> matches <code>acce</code> , <code>accce</code> , etc.
<code>{n,m}</code>	Matches from $n$ to $m$ occurrences of the previous character	<code>ac{2,4}e</code> matches <code>acce</code> , <code>accce</code> , and <code>acccee</code> .

Literal values of metacharacters must be quoted using `\`



# The Dot Metacharacter

The dot `.` is a metacharacter that matches one occurrence of any character except a new line

`a.e` matches the strings *ale* and *ace* in:

*The aerial acceleration alerted the ace pilot*

as well as *age*, *ape*, *are*, *ate*, *awe*, *axe*, or *aae*, *aAe*, *abe*, *aBe*, *a1e*, etc.

`.*` matches any string of characters until we encounter a new line.





# The Longest Match

The previous slide does not tell about the match strategy.  
Consider the string *aabbc* and the regular expression *a+b\**  
By default the match engine is greedy: It matches as early and as many characters as possible and the result is *aabb*  
Sometimes a problem. Consider the regular expression *<b>.\*</b>* and the phrase

*They match <b>as early</b> and <b>as many</b> characters as they can.*

It is possible to use a lazy strategy with the *\*?* metacharacter instead:  
*<b>.\*?</b>* and have the result:

*They match <b>as early</b> and <b>as many</b> characters as they can.*



# Character Classes

[...] matches any character contained in the list.

[^...] matches any character not contained in the list.

[abc] means one occurrence of either a, b, or c

[^abc] means one occurrence of any character that is not an a, b, or c,

[ABCDEFGHIJKLMNOPQRSTUVWXYZ] one upper-case unaccented letter

[0123456789] means one digit.

[0123456789]+\.[0123456789]+ matches decimal numbers.

[Cc]omputer [Ss]cience matches Computer Science,  
computer Science, Computer science, computer science.



# Predefined Character Classes

Expr.	Description	Example
<code>\d</code>	Any digit. Equivalent to <code>[0-9]</code>	<code>A\dC</code> matches <code>A0C</code> , <code>A1C</code> , <code>A2C</code> , <code>A3C</code> etc.
<code>\D</code>	Any nondigit. Equivalent to <code>[^0-9]</code>	
<code>\w</code>	Any word character: letter, digit, or underscore. Equivalent to <code>[a-zA-Z0-9_]</code>	<code>1\w2</code> matches <code>1a2</code> , <code>1A2</code> , <code>1b2</code> , <code>1B2</code> , etc
<code>\W</code>	Any nonword character. Equivalent to <code>[^\w]</code>	
<code>\s</code>	Any white space character: space, tabulation, new line, form feed, etc.	
<code>\S</code>	Any nonwhite space character. Equivalent to <code>[^\s]</code>	



# Nonprintable Symbols or Positions

Char.	Description	Example
^	Matches the start of a line	^ab*c matches ac, abc, abbc, etc. when they are located at the beginning of a new line
\$	Matches the end of a line	ab?c\$ matches ac and abc when they are located at the end of a line
\b	Matches word boundaries	\babc matches abcd but not dabc bcd\b matches abcd but not abcde
\n	Matches a new line	a\nb matches a b
\t	Matches a tabulation	

```
egrep '^[aeiou]*$' myFile
```



# Union and Boolean Operators

Union denoted  $|$ :  $a|b$  means either  $a$  or  $b$ .

Expression  $a|bc$  matches the strings  $a$  and  $bc$  and  $(a|b)c$  matches  $ac$  and  $bc$ ,

Order of precedence:

- 1 Closure and other repetition operator (highest)
- 2 Concatenation, line and word boundaries
- 3 Union (lowest)

$abc^*$  is the set  $ab$ ,  $abc$ ,  $abcc$ ,  $abccc$ , etc.

$(abc)^*$  corresponds to  $abc$ ,  $abcabc$ ,  $abcabcabc$ , etc.



# Python

## Match: m/regex/

```
import regex as re
```

```
line = 'The aerial acceleration alerted the ace pilot'
```

```
match = re.search('ab*c', line)
```

```
match      # <regex.Match object; span=(11, 13), match='ac'>
```

```
match.group() # ac
```

The `re.search()` function stops at the first match.



# Python

Use `findall()` or `finditer()` to return all the matches

**Match:** `m/regex/g`

```
match_list = re.findall('ab*c', line)    # ['ac', 'ac']
```

**Match:** `m/regex/g`

```
match_iter = re.finditer('ab*c', line)
list(match_iter)
# [<regex.Match object; span=(11, 13), match='ac'>,
#  <regex.Match object; span=(36, 38), match='ac'>]
```



# Match Modifiers

Flags that modifies the match operation. These flags are equivalent to Perl's `m/regex/modifiers`.

- Case insensitive: `i`. The instruction `m/regex/i`. In Python, this corresponds to the flag: `re.I`.
- Multiple lines: `m` (`re.M` in Python). `m/regex/m`.
- Single line: `s` (`re.S` in Python). The `/s` modifier makes a dot in the instruction `m/regex/s` match any character, including new lines.

In Python, the modifiers (called flags) are specified as a sequence separated by vertical bars: `|`.





# Python

## Match: m/regex/modifiers

```
text = sys.stdin.read()
match = re.search('^ab*c', text, re.I | re.M) # m/^ab*c/im
if match:
    print('-> ' + match.group())
```



# Python

## **Substitute:** s/regex/replacement/g

```
for line in sys.stdin:
    if re.search('ab+c', line):
        print("Old: " + line, end='')
        # Replaces all the occurrences
        line = re.sub('ab+c', 'ABC', line)    # s/ab+c/ABC/g
        print("New: " + line, end='')
```

## **Substitute:** s/regex/replacement/

If we just want to replace the first occurrence, we use this statement instead:

```
# Replaces the first occurrence
line = re.sub('ab+c', 'ABC', line, 1) # s/ab+c/ABC/
```

# Backreferences

We can store parts of the matched patterns using backreferences. To do this, we surround these parts with parentheses in the regular expression, for instance:

`^(.)(b+)c+`

- The first part, `(.)`, is stored in variable `\1`
- The second, `(b+)`, is stored in `\2`, and so on.

## Backreferences

```
line = 'abbbbcdeeeef'
match = re.search('^(.)(b+)c+', line)
match.group(1)           # 'a'
match.group(2)           # 'bbb'
```



# Python

## Back references

The instruction `m/(.)\1\1/` matches sequences of three identical characters:

```
line = 'abbbcdeeeef'
match = re.search(r'(.)\1\1', line)
match.group(1)          # 'b'
```

We need to use a raw string and the `r` prefix to encode the regex in `search()`, otherwise `\1` would be interpreted as an octal number

## Substitutions

```
s/(.)\1\1/***/g
re.sub(r'(.)\1\1', '***', 'abbbcdeeeef') # 'a***cd***f'
```

# Python

## Multiple back references

Python can create as many buffers as we need: \1, \2, \3, etc. Outside the regular expression, the \<digit> reference is returned by `group(<digit>)`: `match_object.group(1)`, `match_object.group(2)`, `match_object.group(3)`, etc.

## Multiple back references

```
m/\$ *([0-9]+)\.?([0-9]*)/
```

```
price = "We'll buy it for $72.40"
```

```
match = re.search(r'\$ *([0-9]+)\.?([0-9]*)', price)
```

```
match.group()           # '$72.40' The entire match
```

```
match.group(1)          # '72' The first group
```

```
match.group(2)          # '40' The second group
```

# Python

## Substitutions

```
s/\$ *([0-9]+)\.?([0-9]*)/\1 dollars and \2 cents/g

price = "We'll buy it for $72.40"
re.sub(r'\$ *([0-9]+)\.?([0-9]*)',
      r'\1 dollars and \2 cents', price)
# We'll buy it for 72 dollars and 40 cents
```



# Python

## Match objects

- `match_object.group()` or `match_object.group(0)` return the entire match;
- `match_object.group(n)` returns the *n*th parenthesized subgroup.

In addition, the `match_object.groups()` returns a tuple with all the groups and the `match_object.string` instance variable contains the input string.

```
price = "We'll buy it for $72.40"
match = re.search(r'\$ *([0-9]+)\.?([0-9]*)', price)
match.string           # We'll buy it for $72.40
match.groups()         # ('72', '40')
```



# Python

## Match objects

We extract the indices of the matched substrings with the functions:

```
match_object.start([group])
```

```
match_object.end([group])
```

```
line = """Tell me, O muse, of that ingenious hero  
    who travelled far and wide after he had sacked  
    the famous town of Troy."""
```

```
match = re.search(',.*', line, re.S)
```

```
line[0:match.start()]           # 'Tell me'
```

```
line[match.start():match.end()] # ', O muse,'
```

```
line[match.end():]             # 'of that ingenious hero
```

```
    # who travelled far and wide after he had sacked
```

```
    # the famous town of Troy.'
```



# A Regex to Find Concordances

To print concordances, we need to write a regex that matches the pattern as well as a left and right context.

For instance *Nils Holgersson* with a context of 15 characters:

```
.{0,15}Nils Holgersson.{0,15}
```

Ideally, we would pass pattern and width as parameters:

```
pattern = 'Nils Holgersson'  
width = 15  
'.{0,width}pattern.{0,width}'
```



# format()

`str.format()` provides variable substitutions as in:

```
begin = 'my'
'{} string {}'.format(begin, 'is empty')
                                # 'my string is empty'
```

`format()` has many options like reordering the arguments through indices:

```
begin = 'my'
'{1} string {0}'.format('is empty', begin)
                                # 'my string is empty'
```

If the input string contains braces, we escape them by doubling them: `{ {` for a literal `{` and `} }` for `}`.

```
('.{0,{width}}{pattern}.{0,{width}}'
 .format(pattern=pattern, width=width))
```



# Concordances in Python

```
[file_name, pattern, width] = sys.argv[1:]
try:
    text = open(file_name).read()
except:
    print('Could not open file', file_name)
    exit(0)

# spaces match tabs and newlines
pattern = re.sub(' ', r'\s+', pattern)
# Replaces newlines with spaces in the text
text = re.sub(r'\s+', ' ', text)
concordance = ('({0,{width}}){pattern}.({0,{width}})'.
               .format(pattern=pattern, width=width))
for match in re.finditer(concordance, text):
    print(match.group(1))
```



# Approximate String Matching

A set of edit operations that transforms a source string into a target string: copy, substitution, insertion, deletion, reversal (or transposition). Edits for *acress* from Kernighan et al. (1990).

Type	Correction	Source	Target	Position	Operation
acress	actress	—	t	2	Deletion
acress	cress	a	—	0	Insertion
acress	caress	ac	ca	0	Transposition
acress	access	r	c	2	Substitution
acress	across	e	o	3	Substitution
acress	acres	s	—	4	Insertion
acress	acres	s	—	5	Insertion



# Building a Spell Checker

Spell checkers use a dictionary and a set of transformations to suggest corrections to misspelled words in a text.

Dictionaries are collected from well-written texts: novels, newspapers, etc.

- Given a word in a text not in the dictionary, the spell checker generates all the transformations of this word.
- If we allow only one edit operation on a source string of length  $n$ , and if we consider an alphabet of 26 unaccented letters,
  - the deletion will generate  $n$  new strings;
  - the insertion,  $(n + 1) \times 26$  strings;
  - the substitution,  $n \times 25$ ; and
  - the transposition,  $n - 1$  new strings.
- The spell checker keeps the transformations that are in the dictionary and orders them by frequency to suggest the correct word.

For an implementation, see <http://norvig.com/spell-correct.html>



# Building a Spell Checker

```
freq('acres') = 36.  
freq('caress') = 3.  
freq('cress') = false.  
freq('actress') = 7.  
freq('access') = 56.  
freq('across') = 222.
```



# Distance between $ab$ and $cb$



Edit distances measure the similarity between strings.

Let us align

a	b	Source
c	b	Destination

b	2		
c	1		
Start	0	1	2
	Start	a	b



# Minimum Edit Distance

We compute the minimum edit distance using a matrix where the value at position  $(i,j)$  is defined by the recursive formula:

$$\text{edit\_distance}(i,j) = \min \begin{pmatrix} \text{edit\_distance}(i-1,j) + \text{del\_cost} \\ \text{edit\_distance}(i-1,j-1) + \text{subst\_cost} \\ \text{edit\_distance}(i,j-1) + \text{ins\_cost} \end{pmatrix}.$$

where  $\text{edit\_distance}(i,0) = i$  and  $\text{edit\_distance}(0,j) = j$ .





# Edit Operations



Usually,  $del\_cost = ins\_cost = 1$   
 $subst\_cost = 2$  if  $source(i) \neq target(j)$   
 $subst\_cost = 0$  if  $source(i) = target(j)$ .



# Distance between $ab$ and $cb$



Let us align

a	b	Source
c	b	Destination

b	2		
c	1		
Start	0	1	2
	Start	a	b



# Distance between $ab$ and $cb$



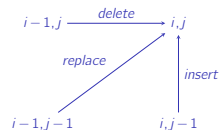
Let us align

a	b	Source
c	b	Destination

b	2		
c	1	2	
Start	0	1	2
	Start	a	b



# Distance between $ab$ and $cb$



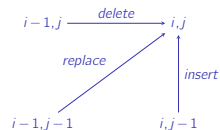
Let us align

a	b	Source
c	b	Destination

b	2	3	
c	1	2	3
Start	0	1	2
	Start	a	b



# Distance between $ab$ and $cb$



Let us align

a	b	Source
c	b	Destination

b	2	3	<b>2</b>
c	1	2	3
Start	0	1	2
Start	a	b	



# Distance between *language* and *lineage*

---

e	7								
g	6								
a	5								
e	4								
n	3								
i	2								
l	1								
Start	0	1	2	3	4	5	6	7	8
	Start	l	a	n	g	u	a	g	e

---



# Distance between *language* and *lineage*

e	7	6	5						
g	6	5	4						
a	5	4	3						
e	4	3	4						
n	3	2	3						
i	2	1	2	3	4	5	6	7	8
l	1	0	1	2	3	4	5	6	7
Start	0	1	2	3	4	5	6	7	8
	Start	l	a	n	g	u	a	g	e



# Distance between *language* and *lineage*

e	7	6	5	6	5	6	7	6	<b>5</b>
g	6	5	4	5	4	5	6	5	6
a	5	4	3	4	5	6	5	6	7
e	4	3	4	3	4	5	6	7	6
n	3	2	3	2	3	4	5	6	7
i	2	1	2	3	4	5	6	7	8
l	1	0	1	2	3	4	5	6	7
Start	0	1	2	3	4	5	6	7	8
	Start	l	a	n	g	u	a	g	e





# Python Code

```
[source, target] = sys.argv[1:]

length_s = len(source) + 1
length_t = len(target) + 1

# Initialize first row and column
table = [None] * length_s

for i in range(length_s):
    table[i] = [None] * length_t
    table[i][0] = i
for j in range(length_t):
    table[0][j] = j
```



# Python Code

```
# Fills the table. Start index of rows and columns is 1
for i in range(1, length_s):
    for j in range(1, length_t):
        # Is it a copy or a substitution?
        cost = 0 if source[i - 1] == target[j - 1] else 2
        # Computes the minimum
        minimum = table[i - 1][j - 1] + cost
        if minimum > table[i][j - 1] + 1:
            minimum = table[i][j - 1] + 1
        if minimum > table[i - 1][j] + 1:
            minimum = table[i - 1][j] + 1
        table[i][j] = minimum

print('Minimum distance: ', table[length_s - 1][length_t - 1])
```



# Distance between *language* and *lineage*

	First alignment	Third alignment
Without epsilon symbols	l a n g u a g e	l a n g u a g e
	/ / /	/ / /
	l i n e a g e	l i n e a g e
With epsilon symbols	l a n g u a g e	l a n g u ε a g e
	l i n e ε a g e	l i n ε ε e a g e

