Language Technology http://cs.lth.se/edan20/

Chapter 2: Corpus Processing Tools

Pierre Nugues

Pierre.Nugues@cs.lth.se
http://cs.lth.se/pierre_nugues/

August 29 and September 1st, 2022



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
Most frequent words in a collection	the	de	der
of contemporary running texts	of	<i>le</i> (article)	die
	to	<i>la</i> (article)	und
	in	et	in
	and	les	des
Most frequent words in Genesis	and	et	und
	the	de	die
	of	la	der
	his	à	100
	he	il 🥞	**************************************

Characteristics of Current Corpora

Big:

- The Bank of English (Collins and U Birmingham) from the 1990s has 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
You say	Strong tea	Thé fort	Schmales Gesicht
	Powerful computer	Ordinateur puissant	Enge Kleidung
You don't	Strong computer	Thé puissant	Schmale Kleidung
say	Powerful tea	Ordinateur fort	Enges Gesicht



Word Preferences

	Strong w			Powerful w	
strong w	powerful w	W	strong w	powerful w	W
161	0	showing	1	32	than
175	2	support	1	32	figure
106	0	support 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:

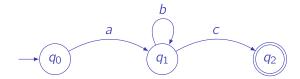
- Translation
- Information and knowledge extraction
- Question answering from textual sources



Finite-State Automata

A flexible to tool to search and process text

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





FSA

Mathematically defined by

- Q a finite number of states;
- \bullet Σ a finite set of symbols or characters: the input alphabet;
- q₀ a start state,
- F a set of final states $F \subseteq Q$
- δ a transition function $Q \times \Sigma \to Q$ 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

Constant regular expressions:

Pattern	String
regular	A section on regular expressions
the	The book of the life

Metacharacters like *, where the automaton above is described by the regex ab*c
Searches and processes text

\$ 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.

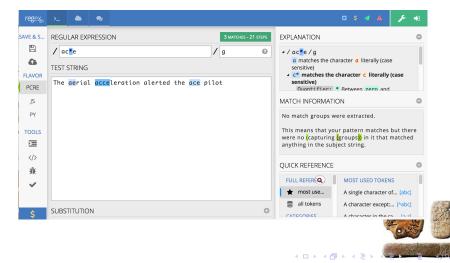
They are often abridged using the Perl regex notations where:

- 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.



Metacharacters

Chars	Descriptions	Examples	
*	Matches any number of occur-	ac*e matches strings ae, ace,	
	rences of the previous charac-	acce, accce, etc. as in "The	
	ter – <mark>zero or more</mark>	aerial acceleration alerted the	
		ace pilot"	
?	Matches at most one occur-	ac?e matches ae and ace as in	
	rence of the previous character	"The <u>ae</u> rial acceleration alerted	
	zero or one	the <u>ace</u> pilot"	
+	Matches <mark>one or more</mark> occur-	ac+e matches ace, acce,	
	rences of the previous charac-	accce, etc. as in as in "The	
	ter	aerial acceleration alerted the	
		ace pilot"	

Metacharacters

Chars	Descriptions	Examples
{n}	Matches exactly <i>n</i> occurrences	ac{2}e matches acce as in
	of the previous character	"The aerial <u>acce</u> leration alerted
		the ace pilot"
{n,}	Matches <mark>n or more</mark> occurrences	ac{2,}e matches acce,
	of the previous character	accce, etc.
{n,m}	Matches from <i>n</i> to <i>m</i> occur-	ac{2,4}e matches acce,
	rences of the previous charac-	accce, and acccce.
	ter	

Literal values of metacharacters must be quoted using \

e.g., .+<\/b>



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 <u>ale</u>rted the <u>ace</u> 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 .* and the phrase

They match $\langle b \rangle$ as early $\langle b \rangle$ and $\langle b \rangle$ as many $\langle b \rangle$ characters as they can.

It is possible to use a lazy strategy with the *? metacharacter instead: .*? and have the result:

They match $\langle b \rangle$ as early $\langle b \rangle$ and $\langle b \rangle$ as many $\langle b \rangle$ 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
- [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
\d	Any digit. Equivalent to [0-9]	A\dC matches A0C, A1C,
		A2C, A3C etc.
\D	Any nondigit. Equivalent to	
	[^0-9]	
\w	Any <mark>word character</mark> : letter,	1\w2 matches 1a2, 1A2, 1b2,
Not recommended	digit, or underscore. Equiva-	1B2, etc
	lent to [a-zA-Z0-9_]	
\W	Any nonword character.	
	Equivalent to [^\w]	
\s	Any white space character:	
	space, tabulation, new line,	Recommended to use
	form feed, etc.	
\S	Any nonwhite space character.	
	Equivalent to [^\s]	

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 <mark>end of a line</mark>	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 <mark>new line</mark>	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:

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

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

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



```
Match: m/regex/
```

```
import regex as re
line = 'The aerial acceleration alerted the ace pilot'
match = re.search('ab*c', line)
            # <regex.Match object; span=(11, 13), match='ac'>
match
                                           Start and end index
match.group() # ac
                                           of the match
                                                              value
```

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

Language Technology http://cs.lth.se/edan20/



Match: m/regex/g

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



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: |



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())
```



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

"The result"

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:

backreferences

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

Backreferences

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



Back references

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

```
line = 'abbbcdeeef'
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', '***', 'abbbcdeeef') # 'a***cd***f'
```



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

Substitutions



Match objects

- match_object.group() or match_object.group(0) return the entire match;
- match_object.group(n) returns the nth parenthetized 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.



Match objects

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

```
match_object.start([group]) start index
match_object.end([group]) End index
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()] # ', 0 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:
```

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

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).

Туро	Correction	Source	Target	Position	Operation
acress	actress	_	t	2	Deletion
acress	cress	a	_	0	Insertion
acress	caress	ac	ca	0	Transposition
acress	access	r	С	2	Substitution
acress	across	е	0	3	Substitution
acress	acres	S	_	4	Insertion
acress	acres	S	_	5	Insertion
					A CONTRACTOR OF THE PARTY OF TH

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.

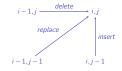
- Given a word in a text not in the dictionary, the spell checker generates all the transformations of this word.
- If we allow only one edit operation on a source string of length *n*, and if we consider an alphabet of 26 unaccented letters,
 - the deletion will generate *n* new strings;
 - the insertion, $(n+1) \times 26$ 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 co

For an implementation, see http://norvig.com/spell-correct_htm

Building a Spell Checker

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





Edit distances measure the similarity between strings.

Let us align
$$\begin{array}{cccc} a & b & Source \\ \hline c & b & Destination \end{array}$$

b	2		
С	1		
Start	0	1	2
	Start	a	b



Minimum Edit Distance

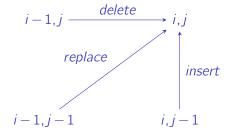
We compute the minimum edit distance using a matrix where the value at position (i,j) is defined by the recursive formula:

$$edit_distance(i,j) = \min \left(\begin{array}{c} edit_distance(i-1,j) + del_cost \\ edit_distance(i-1,j-1) + subst_cost \\ edit_distance(i,j-1) + ins_cost \\ \end{array} \right).$$

where $edit_distance(i,0) = i$ and $edit_distance(0,j) = j$.



Edit Operations

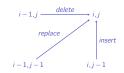


```
Usually, del\_cost = ins\_cost = 1

subst\_cost = 2 if source(i) \neq target(j)

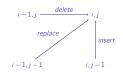
subst\_cost = 0 if source(i) = target(j).
```





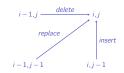
b	2		
С	1		
Start	0	1	2
	Start	a	b





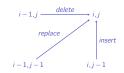
b	2		
С	1	2	
Start	0	1	2
	Start	a	b





b	2	3	
С	1	2	3
Start	0	1	2
	Start	a	b





b	2	3	2
С	1	2	3
Start	0	1	2
	Start	a	b



```
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
```



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



е	7	6	5	6	5	6	7	6	5
g	6	5	4	5	4	5	6	5	6
a	5	4	3	4	5	6	5	6	7
е	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
1	1	0	1	2	3	4	5	6	7
Start	0	1	2	3	4	5	6	7	8
	Start		a	n	g	u	a	g	е



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][i] = i
```



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][i - 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_s - 1]

	First alignment	Third alignment			
Without epsilon symbols		1 1 1 7 7 7			
	l i n e a g e	lineage			
	language	langu ε age			
With epsilon symbols					
	line ε age	lin ε ε eage			

