Natural Language Processing (NLP)

Basic Text Processing

Based on Jurafsky & Martin (2018), Chapter 2

Katrien Beuls

Artificial Intelligence Laboratory
Vrije Universiteit Brussel

VRIJE
UNIVERSITEIT
BRUSSEL

BASIC TEXT PROCESSING

ELIZA

```
User:  You are like my father in some ways.
ELIZA: WHAT RESEMBLANCE DO YOU SEE
User:  You are not very aggressive but I think you don't want me to
       notice that.
ELIZA: WHAT MAKES YOU THINK I AM NOT VERY AGGRESSIVE
User:  You don't argue with me.
ELIZA: WHY DO YOU THINK I DON'T ARGUE WITH YOU
User:  You are afraid of me.
ELIZA: DOES IT PLEASE YOU TO BELIEVE I AM AFRAID OF YOU
```

Weizenbaum (1966)

BASIC TEXT PROCESSING

REGULAR EXPRESSIONS

- A formal language for specifying text strings
- How can we search for any of these?
  - woodchuck
  - woodchucks
  - Woodchuck
  - Woodchucks

REGULAR EXPRESSIONS

DISJUNCTIONS

- Letters inside square brackets: `[]`

Pattern	Matches
`[wW]oodchuck`	Woodchuck, woodchuck
`[1234567890]`	Any digit

- Ranges: `[A-Z]`

Pattern	Matches	
`[A-Z]`	An upper case letter	`Drenched Blossoms`
`[a-z]`	A lower case letter	`my beans were impatient`
`[0-9]`	A single digit	`Chapter 1: Down ...`

REGULAR EXPRESSIONS

NEGATION IN DISJUNCTION

- Negations: `[^Ss]`

Pattern	Matches	
`[^A-Z]`	Not an upper case letter	`Oyfn pripetchik`
`[^Ss]`	Neither 'S' nor 's'	`I have no reason`
`[^\.]`	Not a period	`our resident Djinn`
`a^b`	The pattern 'a^b'	`look up a^b now`

REGULAR EXPRESSIONS

MORE DISJUNCTION

- Woodchucks is another name for groundhog!
- The pipe | for disjunction

Pattern	Matches
`groundhog|woodchuck`	groundhog woodchuck
`gupp(y|ies)`	guppy guppies
`a|b|c`	= [abc]
`[gG]roundhog|[Ww]oodchuck`	

REGULAR EXPRESSIONS

? * + .

Pattern	Matches	
`colou?r`	Optional previous char	color colour
`oo*h!`	0 or more of previous char	oh! ooh! oooh! ooooh!
`o+h!`	1 or more of previous char	oh! ooh! oooh! ooooh!
`baa+`		baa baaa baaaa baaaaa
`beg.n`		begin begun began beg3n

REGULAR EXPRESSIONS

ANCHORS ^ $

Pattern	Matches	
`^[A-Z]`	Palo Alto	
`^[^A-Za-z]`	1 “Hello”	
`\.$`	The end.	
`.$`	The end?	The end!

REGULAR EXPRESSIONS

EXERCISE

- Find all instances of the word “the” in a text.
  - `/the/`
    Misses capitalised examples
  - `/[tT]he/`
    Incorrectly returns `other` or `theology`
  - `/[^a-zA-Z][tT]he[^a-zA-Z]/`
    Does not return “the” when it begins a line
  - `/(^|[^a-zA-Z])[tT]he([^a-zA-Z]|$)/`

REGULAR EXPRESSIONS

OPERATOR PRECEDENCE HIERARCHY

- From highest to lowest precedence:

Parenthesis	`()`
Counters	`*` `+` `?` `{}`
Sequences and anchors	`the` `^my` `end$`
Disjunction	`|`

- **Counters > Sequences**: `/the*/` matches *theeeee* but not *thethe*
- **Sequences > Disjunction**: `/the|any/` matches *the* or *any* but not *theny*

REGULAR EXPRESSIONS

SUBSTITUTIONS

- Substitution operator `s/regexp1/pattern/` used in Unix commands like `vim` or `sed` allows a string characterized by a regular expression to be replaced by another string:

  ```
  s/colour/color
  ```

- Referring to a subpart of the string matching the first pattern: e.g. put angle brackets around all integers in a text:

  ```
  s/[0-9]+/<\1>/
  ```

REGULAR EXPRESSIONS

CAPTURE GROUPS

```
/the (.*)er they were, the \1er they will be/
```

- Parenthesis used for storing a pattern in memory
  = **a capture group**
  - Resulting match is stored in a numbered **register**
  - If you match two different sets of parentheses, `\2` means whatever matched the *second* capture group
    ```
    /the (.*)er they (.*), the \1er we \2 /
    ```
- Use a **non-capturing group** if you don't want to capture the resulting pattern in a register:
  ```
  /((?:some|a few) (people|cats) like some \1/
  ```

REGULAR EXPRESSIONS

ELIZA OR A SIMPLE CHATBOT

- Works by having a cascade of (ranked) regular expression substitutions
  - Input lines are first uppercased
  - First substitutions change all instances of *MY* to *YOUR* and *I'M* to *YOU ARE*
  - The next set of substitutions matches and replaces other patterns in the input

```
s/.* I'M (depressed|sad) .*/I AM SORRY TO HEAR YOU ARE \1/
s/.* I AM (depressed|sad) .*/WHY DO YOU THINK YOU ARE \1/
s/.* all .*/IN WHAT WAY/
s/.* always .*/CAN YOU THINK OF A SPECIFIC EXAMPLE/
```

WORDS

WHAT COUNTS AS A WORD?

- How many words are in the following Brown sentence?
  - *He stepped out into the hall, was delighted to encounter a water brother.*
  - 13 words (15 with punctuation marks)
- How many words are in the following utterance from the Switchboard corpus?
  - *I do uh main- mainly business data processing*
  - Two disfluencies: a fragment (*main-*) and a filled pause (*uh*)

WORDS

WHAT COUNTS AS A WORD?

- How about inflected forms like *cats* versus *cat*?
  - They have the same lemma *cat* but different wordforms
  - A **lemma** is a set of lexical forms having the same stem, the same major part-of-speech, and the same word sense
  - The **wordform** is the fully inflected or derived form of the word

WORDS

HOW MANY WORDS ARE THERE IN ENGLISH?

- **Types** are the number of distinct words in a corpus
  - If the set of words in the vocabulary is V, the number of types is the vocabulary size $|V|$
- **Tokens** are the total number N of running words
- If we ignore punctuation, the following Brown sentence has 16 tokens and 14 types:
  - *They picknicked by the pool, then lay back on the grass and looked at the stars.*

WORDS

POPULAR ENGLISH LANGUAGE CORPORA

Corpus	**Tokens = N**	**Types = $\|V\|$**
Shakespeare	884 thousand	31 thousand
Brown corpus	1 million	38 thousand
Switchboard telephone conversations	2.4 million	20 thousand
COCA	440 million	2 million
Google N-grams	1 trillion	13 million

The relationship between the number of types and number of tokens is called **Herdan's Law** (Herdan, 1960) or **Heap's Law** (Heaps, 1978), where k and β are positive constants and $0 < \beta < 1$:

$$|V| = kN^{\beta}$$

WORDS

DICTIONARY ENTRIES

- Look at the **number of lemmas** instead of wordform types
- **Dictionaries** can help in giving lemma counts
  - Dictionary entries or boldface forms are a very rough upper bound on the number of lemmas
  - The 1989 edition of the Oxford English Dictionary had 615,000 entries

TEXT NORMALIZATION

FIRST NLP TASK

At least three tasks are commonly applied as part of any normalization process:

1. Segmenting/tokenizing words from running text
2. Normalizing word formats
3. Segmenting sentences in running text

TEXT NORMALIZATION

WORD TOKENIZATION

Main challenges:

- Break off **punctuation** as a separate token but preserve it when it occurs word internally (*Ph.D.*, *AT&T*, ...)
- Keep **special characters** and **numbers** in prices ($45.55) and dates (15/02/2019)
- Expand **clitic** contractions that are marked by apostrophes (*we're* $\rightarrow$ *we are*)
- Tokenize **multiword expressions** like *New York* or *rock 'n' roll* as a single token

TEXT NORMALIZATION

PENN TREEBANK TOKENIZATION

Penn Treebank tokenization standard separates out clitics (*doesn't* becomes *does* plus *n't*), keeps hyphenated words together, and separates out all punctuation:

Input: "The San Francisco-based restaurant," they said, "doesn't charge $10".

Output:

"	The	San	Francisco-based	restaurant	,	"	they
said	,	"	does	n't	charge	$	10	"	.

TEXT NORMALIZATION

NORMALIZING TOKENS

Choosing a **single normalized form** for words with multiple forms such as *USA* and *US*.

- Valuable for **information retrieval** if you want to query for *US* to match a document that has *USA*
- In **information extraction** we might want to extract coherent information that is consistent across differently-spelled instances

TEXT NORMALIZATION

CASE FOLDING

Case folding is another kind of normalization

- For tasks like speech recognition and information retrieval, everything is mapped to lower case
- For sentiment analysis and other text classification tasks, information extraction and machine translation, case is quite helpful and case folding is generally not done

TEXT NORMALIZATION

EFFICIENCY

Because tokenization needs to be run before any other language processing, it is important to be **very fast**.

- **Deterministic algorithms based on regular expressions** compiled into very efficient finite state automata
- Carefully designed deterministic algorithms can deal with the **ambiguities** that arise (e.g. the apostrophe)

TEXT NORMALIZATION

COLLAPSING WORDS

- **Lemmatization** is the task of determining that two words have the same root, despite their surface differences
- How is lemmatization done?
  - Most sophisticated methods for lemmatization involve complete **morphological parsing** of the word
  - Two broad classes of morphemes can be distinguished:
    1. **Stems** - the central morpheme of the word, supplying the main meaning
    2. **Affixes** - adding “additional” meanings of various kinds

TEXT NORMALIZATION

COLLAPSING WORDS

- **Stemming** is a simpler but cruder method, which mainly consists of chopping off word-final affixes
- One of the most widely used stemming algorithms for English is **the Porter stemmer** (1980)

```
This was not the map we found in Billy Bones’s chest, but
an accurate copy, complete in all things-names and heights
and soundings-with the single exception of the red crosses
and the written notes.
```

```
Thi wa not the map we found in Billi Bone s chest but an
accur copi complet in all thing name and height and sound
with the singl except of the red cross and the written note
```

TEXT NORMALIZATION

COLLAPSING WORDS

- The Porter stemmer algorithm is based on a series of rewrite rules in series, as a **cascade**

ATIONAL	$\rightarrow$	ATE	(e.g. relational $\rightarrow$ relate)
ING	$\rightarrow$	ϵ	if stem contains vowel (e.g., motoring $\rightarrow$ motor)
SSES	$\rightarrow$	SS	(e.g., grasses $\rightarrow$ grass)

- Detailed rule lists for the Porter stemmer can be found on Martin Porter's homepage

TEXT NORMALIZATION

SENTENCE SEGMENTATION

Sentence segmentation is another important step in text processing

- The most useful cues are **punctuation** (periods, question marks, exclamation points)
- Periods can be ambiguous: *Mr.* or *Inc.*

In general, **sentence tokenization methods** work by building a **binary classifier** that decides if a period is a part of the word or is a sentence-boundary marker

MINIMUM EDIT DISTANCE

STRING SIMILARITY

- Calculating the similarity between two strings is useful in many NLP tasks, such as spelling correction or coreference resolution
- The **minimum edit distance** between two strings is defined as the minimum number of editing operations (operations like insertion, deletion, substitution) needed to transform one string into another

MINIMUM EDIT DISTANCE

EXAMPLE ALIGNMENT

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s   i s
```

- The gap between *intention* and *execution* is 5

MINIMUM EDIT DISTANCE

LEVENSHTEIN DISTANCE

- We can also assign a particular **cost** or **weight** to each of these operations
- **Levenshtein distance**: Each insertion or deletion has a cost of 1 and substitutions are not allowed
  - (This is equivalent to allowing substitution, but giving each substitution a cost of 2 since any substitution can be represented by one insertion and one deletion)
- Using this metric, the Levenshtein distance between *intention* and *execution* is 8

MINIMUM EDIT DISTANCE

ALGORITHM

- **How do we find the minimum edit distance?**
  - We can think of this as a search task, in which we are searching for **the shortest path - a sequence of edits - from one string to another**
- The space of all possible edits is enormous, so we can't search naively
- However, lots of distinct edit paths will end up in the same state (string), so rather than recomputing all those paths, we could just remember the shortest path to a state each time we saw it

MINIMUM EDIT DISTANCE

DYNAMIC PROGRAMMING

- **Dynamic programming** is the name for a class of algorithms, first introduced by Bellman (1957), that apply **a table-driven method to solve problems by combining solutions to sub-problems**
- Some of the most commonly used algorithms in NLP make use of dynamic programming, such as the Viterbi algorithm and the CKY algorithm for parsing

MINIMUM EDIT DISTANCE

SHORTEST PATH

```
i n t e n t i o n
                    ← delete i
n t e n t i o n
                    ← substitute n by e
e t e n t i o n
                    ← substitute t by x
e x e n t i o n
                    ← insert u
e x e n u t i o n
                    ← substitute n by c
e x e c u t i o n
```

MINIMUM EDIT DISTANCE

ALGORITHM

```
function MIN-EDIT-DISTANCE(source, target) returns min-distance

  n ← LENGTH(source)
  m ← LENGTH(target)
  Create a distance matrix distance[n+1,m+1]

  # Initialization: the zeroth row and column is the distance from the empty string
        D[0,0] = 0
        for each row i from 1 to n do
            D[i,0] ← D[i-1,0] + del-cost(source[i])
        for each column j from 1 to m do
            D[0,j] ← D[0, j-1] + ins-cost(target[j])

  # Recurrence relation:
  for each row i from 1 to n do
        for each column j from 1 to m do
           D[i, j] ← MIN( D[i−1, j] + del-cost(source[i]),
                          D[i−1, j−1] + sub-cost(source[i], target[j]),
                          D[i, j−1] + ins-cost(target[j]))
  # Termination
  return D[n,m]
```

MINIMUM EDIT DISTANCE

THE EDIT DISTANCE MATRIX

Src\Tar	#	e	x	e	c	u	t	i	o	n
#	0	1	2	3	4	5	6	7	8	9
i	1	2	3	4	5	6	7	6	7	8
n	2	3	4	5	6	7	8	7	8	7
t	3	4	5	6	7	8	7	8	9	8
e	4	3	4	5	6	7	8	9	10	9
n	5	4	5	6	7	8	9	10	11	10
t	6	5	6	7	8	9	8	9	10	11
i	7	6	7	8	9	10	9	8	9	10
o	8	7	8	9	10	11	10	9	8	9
n	9	8	9	10	11	12	11	10	9	8

MINIMUM EDIT DISTANCE

PRODUCING AN ALIGNMENT

	#	e	x	e	c	u	t	i	o	n
#	0	← 1	← 2	← 3	← 4	← 5	← 6	← 7	← 8	← 9
i	↑ **1**	↖←↑ 2	↖←↑ 3	↖←↑ 4	↖←↑ 5	↖←↑ 6	↖←↑ 7	↖ 6	← 7	← 8
n	↑ 2	↖←↑ **3**	↖←↑ 4	↖←↑ 5	↖←↑ 6	↖←↑ 7	↖←↑ 8	↑ 7	↖←↑ 8	↖ 7
t	↑ 3	↖←↑ 4	↖←↑ **5**	↖←↑ 6	↖←↑ 7	↖←↑ 8	↖ 7	←↑ 8	↖←↑ 9	↑ 8
e	↑ 4	↖ 3	← 4	↖← **5**	← **6**	← 7	←↑ 8	↖←↑ 9	↖←↑ 10	↑ 9
n	↑ 5	↑ 4	↖←↑ 5	↖←↑ 6	↖←↑ 7	↖←↑ **8**	↖←↑ 9	↖←↑ 10	↖←↑ 11	↖↑ 10
t	↑ 6	↑ 5	↖←↑ 6	↖←↑ 7	↖←↑ 8	↖←↑ 9	↖ **8**	← 9	← 10	←↑ 11
i	↑ 7	↑ 6	↖←↑ 7	↖←↑ 8	↖←↑ 9	↖←↑ 10	↑ 9	↖ **8**	← 9	← 10
o	↑ 8	↑ 7	↖←↑ 8	↖←↑ 9	↖←↑ 10	↖←↑ 11	↑ 10	↑ 9	↖ **8**	← 9
n	↑ 9	↑ 8	↖←↑ 9	↖←↑ 10	↖←↑ 11	↖←↑ 12	↑ 11	↑ 10	↑ 9	↖ **8**

MINIMUM EDIT DISTANCE

EXTENSIONS

- The algorithm **allows arbitrary weights** on the operations
  - For spelling correction, substitutions are more likely to happen between letters that are next to each other on the keyboard
- The **Viterbi algorithm** is a probabilistic extension of minimum edit distance
  - Viterbi computes the “maximum probability alignment” of one string with another (cf. Ch. 8 on POS tagging)

SUMMARY

- The **regular expression** language is a powerful tool for pattern-matching.
- Basic operations in regular expressions include **concatenation** of symbols, **disjunction** of symbols ([], |, and .), **counters** (*, +, and {n,m}), anchors (ˆ, \$) and **precedence** operators ((,)).
- **Word tokenization and normalization** are generally done by cascades of simple regular expressions substitutions or finite automata.
- The **Porter algorithm** is a simple and efficient way to do **stemming**, stripping off affixes. It does not have high accuracy but may be useful for some tasks.
- The **minimum edit distance** between two strings is the minimum number of operations it takes to edit one into the other. Minimum edit distance can be computed by **dynamic programming**, which also results in an **alignment** of the two strings.