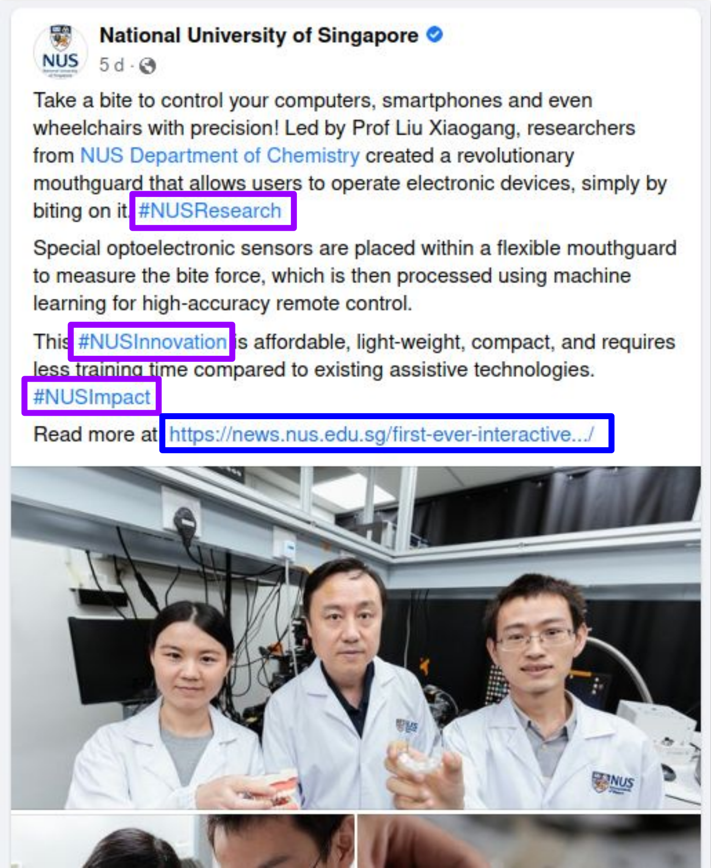
NUS | Computing

National University of Singapore

Natural Language Processing: Foundations

Section 2 — Regular Expressions I (Basic Concepts)

Regular Expressions — Motivating Example



National University of Singapore

5 d ·

Take a bite to control your computers, smartphones and even wheelchairs with precision! Led by Prof Liu Xiaogang, researchers from NUS Department of Chemistry created a revolutionary mouthguard that allows users to operate electronic devices, simply by biting on it. #NUSResearch

Special optoelectronic sensors are placed within a flexible mouthguard to measure the bite force, which is then processed using machine learning for high-accuracy remote control.

This #NUSInnovation is affordable, light-weight, compact, and requires less training time compared to existing assistive technologies. #NUSImpact

Read more at https://news.nus.edu.sg/first-ever-interactive.../

Common task: pattern-based substring matching

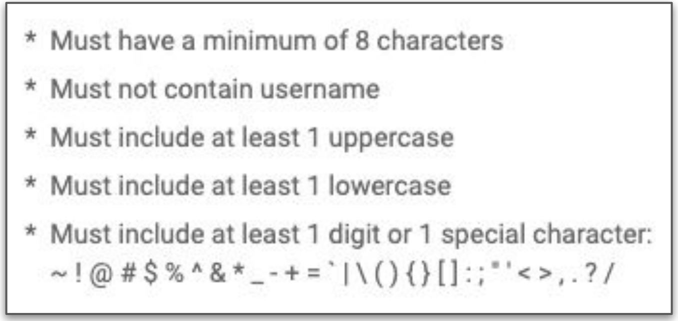
- **Hashtags**, email addresses, **URLs**
- Emoticons, emojis (unicode)
- Dates, phone numbers, IDs
- Postal addresses, lat/long coordinates
- Stock ticker symbols (e.g., "AAPL", "GOOG", "AMZN")
- Brand names (e.g. "Apple", "Google", "Amazon")
- …

➜ **Regular Expression** to rule them all!

Regular Expressions

- Regular Expression — Definition
  - Search pattern used to match character combinations in a string
  - Pattern = sequence of characters

Example: password validation



* Must have a minimum of 8 characters
* Must not contain username
* Must include at least 1 uppercase
* Must include at least 1 lowercase
* Must include at least 1 digit or 1 special character:
  ~ ! @ # $ % ^ & * _ - + = ` | \ ( ) { } [ ] : ; " ' < > , . ? /

- Common applications
  - Parse text documents to find specific character patterns
  - Validate text to ensure it matches predefined patterns
  - Extract, edit, replace, delete substrings matching a pattern

- Two basic search approaches
  - Default: match only first occurrence of pattern
  - Global search: match all occurrences of pattern (assumed in most following examples)

Basic Patterns

- Fixed patterns

`floor` ➜ *My block has 15 floors, and I live on floor 5.*

`5` ➜ *My block has 15 floors, and I live on floor 5.*

`blocks` ➜ *My block has 15 floors, and I live on floor 5.*

- Special characters (metacharacters)

Character	Explanation
`.`	matches any character except line breaks
`^`	match the start of a string
`$`	match the end of a string
`\b`	matches boundary between word and non-word
`|`	matches RegEx either before or after the symbol (e.g., `floor|floors`)

Character Classes

- Character class
  - Defines set of valid characters
  - Enclosed using "`[...]`"
  - Can be negated: "`[^...]`"

`[0-9][0-9]` ➜ *My block has 15 floors, and I live on floor 5.*

(match all sequences of 2 digits)

`[.,;:]` ➜ *My block has 15 floors, and I live on floor 5.*

(match all sequences of length 1 that are either a period, comma, etc.)

`[^a-z]` ➜ *My block has 15 floors, and I live on floor 5.*

(match all sequences of length 1 that are not a lowercase letter)

Predefined Character Classes

- Common character classes with their own shorthand notation (i.e., metacharacters)

Class	Alternative	Explanation
\d	[0-9]	matches any digit
\D	[^0-9]	matches any non-digit
\s	[\n\r\t\f]	matches any whitespace character
\S	[^ \n\r\t\f]	matches any non-whitespace character
\w	[a-zA-Z0-9_]	matches any word character
\W	[^a-zA-Z0-9_]	matches any non-word character

Repetition Patterns

- Very common: patterns with flexible lengths, e.g.:
    - All numbers with more than 2 digits
    - All words with less than 5 characters

- Repetition patterns — metacharacters

Pattern	Explanation
`+`	1 or more occurrences
`*`	0 or more occurrences
`?`	0 or 1 occurrences
`{n}`	exactly `n` occurrences
`{l,u}`	between `l` and `u` occurrences; can be unbounded: `{l,}` or `{,u}`

Repetition Patterns — Examples

`\d{2,}` ➜ *My block has 15 floors, and I live on floor 5.*

(match all numbers with 2 or more digits)

`\d+` ➜ *My block has 15 floors, and I live on floor 5.*

(match all numbers with 1 or more digits)

`\b\w{2,4}\b` ➜ *My block has 15 floors, and I live on floor 5.*

(match words with 2 to 4 characters)

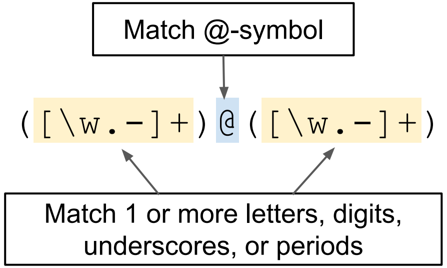
`\b[Ff]loor[s]?\b` ➜ *My block has 15 floors, and I live on floor 5.*

(match occurrences of "floor", either capitalized or not, either in singular or plural)

Groups

- Groups: Organizing patterns into parts
  - Groups are enclosed using "(...)"
  - While whole expression must match, groups are captured individually
    (a match is no longer a string but a tuple of strings, on for each group)
  - Groups can be nested, e.g., (...(...)...((...))...)
    (order of groups depends on the order in which the groups "open")

Send an email to alice@example.org for more information.



Match @-symbol

([\w.-]+)@([\w.-]+)

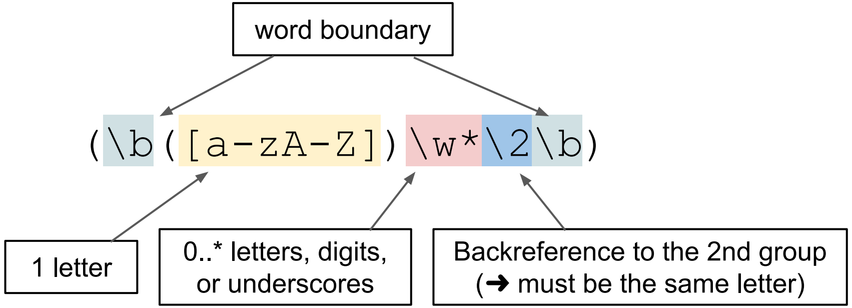
Match 1 or more letters, digits, underscores, or periods

Match:	*user@example.org*
Group #1:	*alice*
Group #2:	*example.org*

Backreferences

- Reference groups within a RegEx
  - Find repeated patterns (see example below)
  - Support only partial replacement of matches

- Example:
  - *"My mom said I need to pass this test."*
  - Goal: Find all words that start and end with the same letter



```
(\b([a-zA-Z])\w*\2\b)
```

word boundary
1 letter
0..* letters, digits, or underscores
Backreference to the 2nd group (➔ must be the same letter)

Match:	*mom*
Group #1:	*mom*
Group #2:	*m*

Match:	*test*
Group #1:	*test*
Group #2:	*t*

Lookarounds

- Special groups — assertions
  - Match like any other group, but do not capture the match
  - 2 types: lookaheads and lookbehinds
  - 2 forms of assertion: positive and negative

	Type	Example
`(?=)`	positive lookahead	`A(?=B)` ➜ finds expr. A but only when followed by expr. B
`(?!)`	negative lookahead	`A(?!B)` ➜ finds expr. A but only when not followed by expr. B
`(?<=)`	positive lookbehind	`(?<=B)A` ➜ finds expr. A but only when preceded by expr. B
`(?<!)`	negative lookbehind	`(?<!B)A` ➜ finds expr. A but only when not preceded by expr. B

Lookarounds — Example

- Positive lookahead
  - *"Paying 10 SGD for 1 kg of chicken seems fair."*
  - Goal: Extract all *kg* values (numbers followed by the unit *kg*)

`\d+(?=\s*kg)` ➜

"Paying 10 SGD for 1 kg of chicken seems fair.
"Paying 10 SGD for 1.5 kg of chicken seems fair.
"Paying 10 SGD for 1,500.00 kg of chicken seems fair.

`[0-9.,]*[0-9]+(?=\s*kg)` ➜

"Paying 10 SGD for 1 kg of chicken seems fair.
"Paying 10 SGD for 1.5 kg of chicken seems fair.
"Paying 10 SGD for 1,500.00 kg of chicken seems fair.

NUS | Computing

National University of Singapore

Natural Language Processing: Foundations

Section 2 — Regular Expressions II (Relationship to FSA)

Relationship to Finite State Automata

- Equivalence
  - Regular Expressions describe **Regular Languages**
    (most restricted type of languages w.r.t Chomsky Hierarchy)
  - Regular Language = language accepted by a FSA

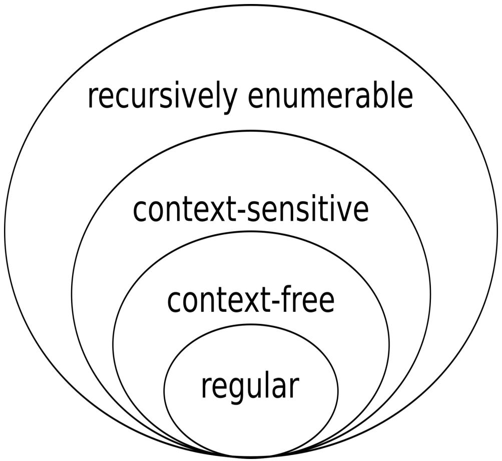
Regular Expression

l(o+l)+

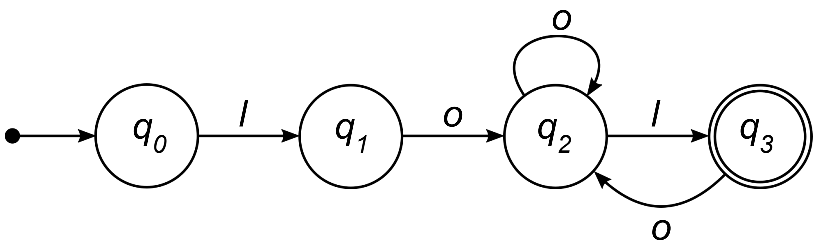
Regular Language

{lol, loool, lolol, looolol, …}

Chomsky Hierarchy
(Source: Wikipedia)



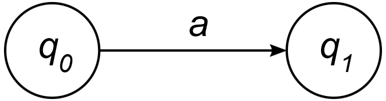
Example: FSA that accepts the Regular Language described by the Regular Expression **l(o+l)+**



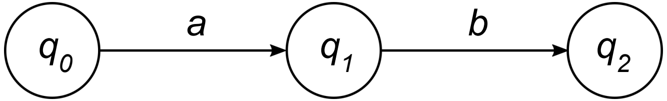
Relationship to Finite State Automata

- Basic equivalences

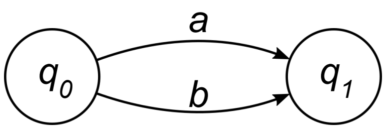
a



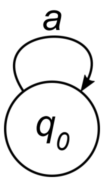
ab



a | b



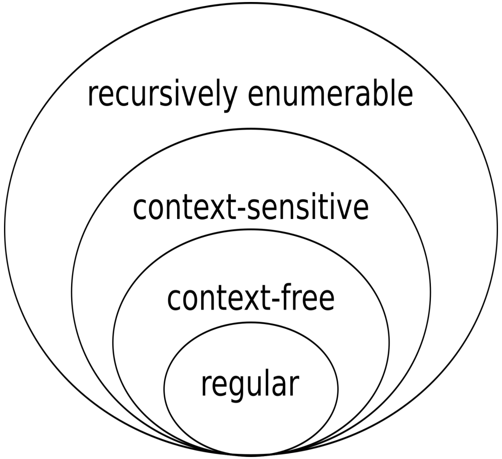
a*



Regular Expressions — Limitations

- Nonregular Languages

$$L = \{0^n 1^n \mid n \in \mathbb{N}\} = \{\epsilon, 01, 0011, 000111, 00001111, ...\}$$



***L* is not a Regular Language** — intuition:

- We need to "remember" an arbitrary number of 0s
- We cannot do this with a finite number of states
- Note the *L* is a subset of all possible palindromes

➜ We cannot write a RegEx that describes *L*!

Important note: Many modern RegEx engines go beyond to limitations of Regular Languages. Such engines implement concepts such recursive references and balancing groups. These implementations allow, for example, to find arbitrary palindromes in a text.

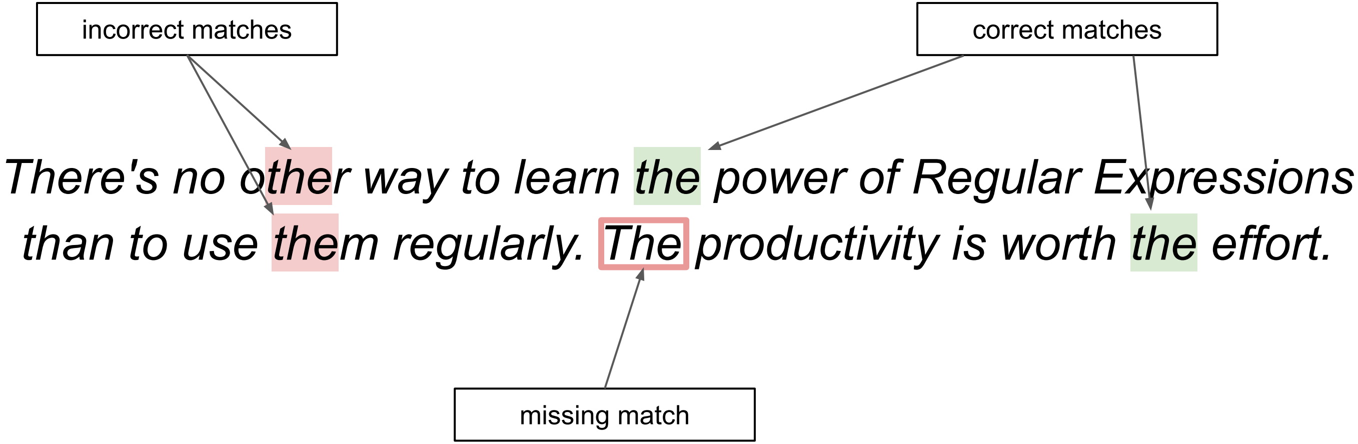
NUS | Computing
National University of Singapore

Natural Language Processing: Foundations

Section 2 — Regular Expressions III (Error Types)

Error Types — What Can Go Wrong

- Example: Find all occurrences of article "the"
  - Naive approach: "`the`" (fixed pattern)



incorrect matches

correct matches

There's no other way to learn the power of Regular Expressions than to use them regularly. The productivity is worth the effort.

missing match

Error Types

- 2 basic types of errors

Matching strings that we should not have matched

(e.g., *other*, *theology*, *weather*, *bathe*, *mother*)

➜ **False Positives**
(Type I Errors)

Not matching things that we should have matched

(e.g., *The*)

➜ **False Negatives**
(Type II Errors)

Error Types — Observations

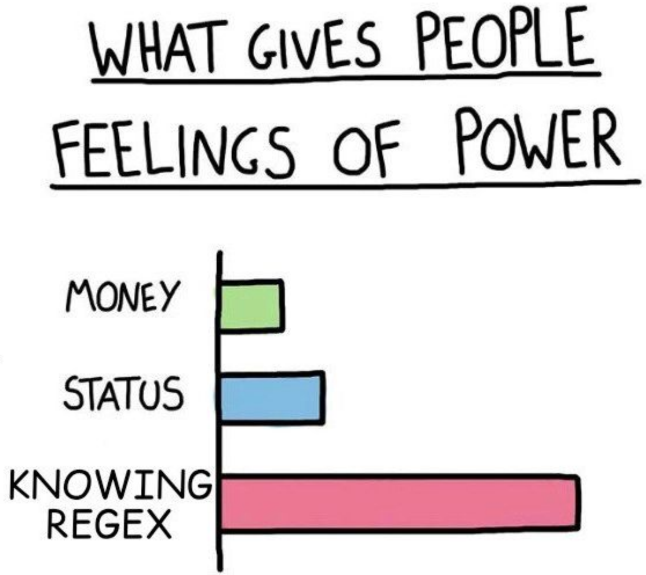
- Many contexts deal with these 2 types of errors, e.g.:
  - Medical testing (e.g., ART test is positive but person is not infected with COVID ➜ false positive)
  - Information retrieval (e.g., a Web search is missing a relevant page ➜ false negative)
  - Document classification (e.g., an abusive tweet has be classified as positive ➜ false positive)

- Reducing errors
  - Both error types are not always equally bad ($\overbrace{\text{infected person tests negative}}^{\textbf{false negative}}$ vs. $\overbrace{\text{healthy person test positive}}^{\textbf{false positive}}$)
  - Reducing False Positives and False Negatives often in conflict
    (reducing False Positives often increases False Negatives, and vice versa)

Regular Expressions — Summary

- Know their powers
  - Extremely useful tool for many (low-level) text processing tasks
    (e.g., data preprocessing, tokenization, normalization)
  - Important skill for anyone working with strings or text
- Know their limitations
  - Regular Expressions represent hard rules
  - Higher-level text processing task generally require statistical models ("soft" rules)

➜ Machine Learning classifiers



NUS | Computing

National University of Singapore

Natural Language Processing: Foundations

Section 2 — Text Preprocessing: From Strings to Words

Text Preprocessing — 2 Main Purposes

(1) Text as string ➜ Text as written Natural Language

Text as sequence of characters ➜ Text as **collection** of **tokens**

collection: set, multiset/bag, sequence

tokens: subwords, words, phrases

Tokens convey meaning more than characters

(2) Raw source text ➜ "proper" input for NLP methods

Text cleaning: remove any kind of noise
(e.g. HTML tags, unicode characters, URLs)

Address challenges associated with Natural Language
(e.g., lower redundancy, lower variability, make it bounded)

Make it easier for NLP methods to identify patterns

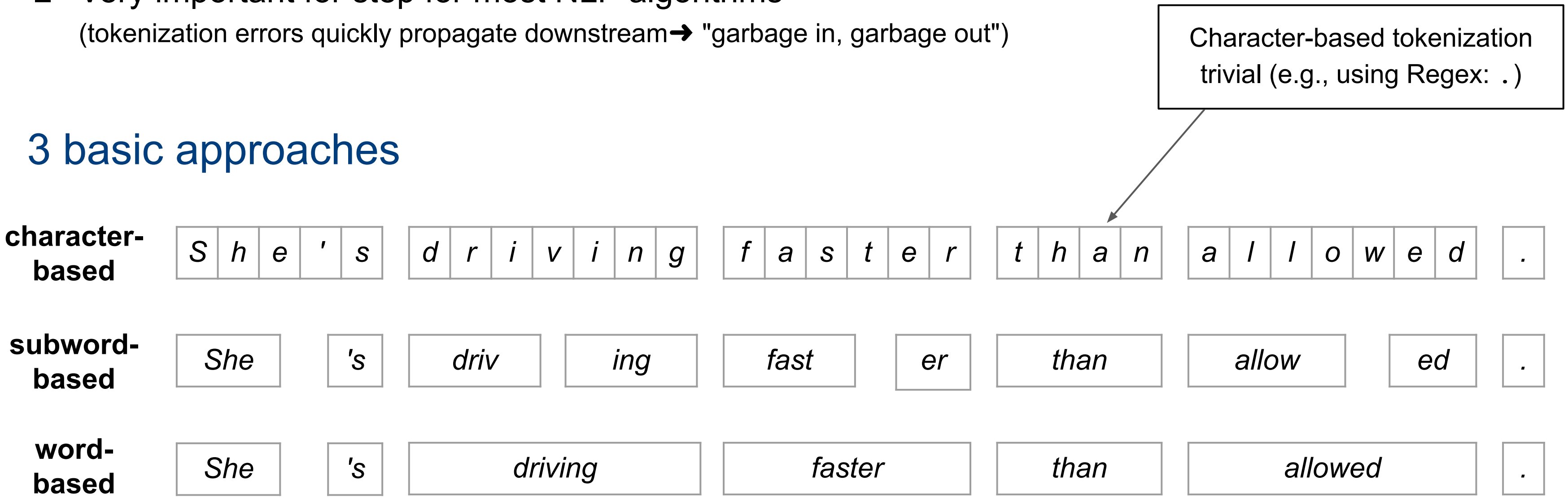
NUS | Computing
National University of Singapore

Natural Language Processing: Foundations

Section 2 — Text Preprocessing I (Tokenization)

Tokenization

- Tokenization: splitting a string into **tokens ➜ vocabulary** (set of all unique tokens)
  - Token = character sequence with semantic meaning
    (typically: words, numbers, punctuation — but may differ depending on applications)
  - Very important for step for most NLP algorithms
    (tokenization errors quickly propagate downstream➜ "garbage in, garbage out")
- 3 basic approaches

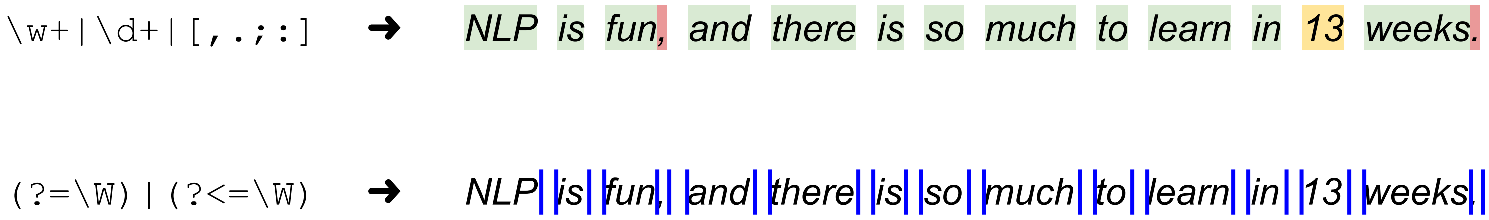


Character-based tokenization trivial (e.g., using Regex: .)

character-based	S h e ' s		d r i v i n g		f a s t e r		t h a n	a l l o w e d		.
subword-based	She	's	driv	ing	fast	er	than	allow	ed	.
word-based	She	's	driving		faster		than	allowed		.

Tokenization — Word-Based

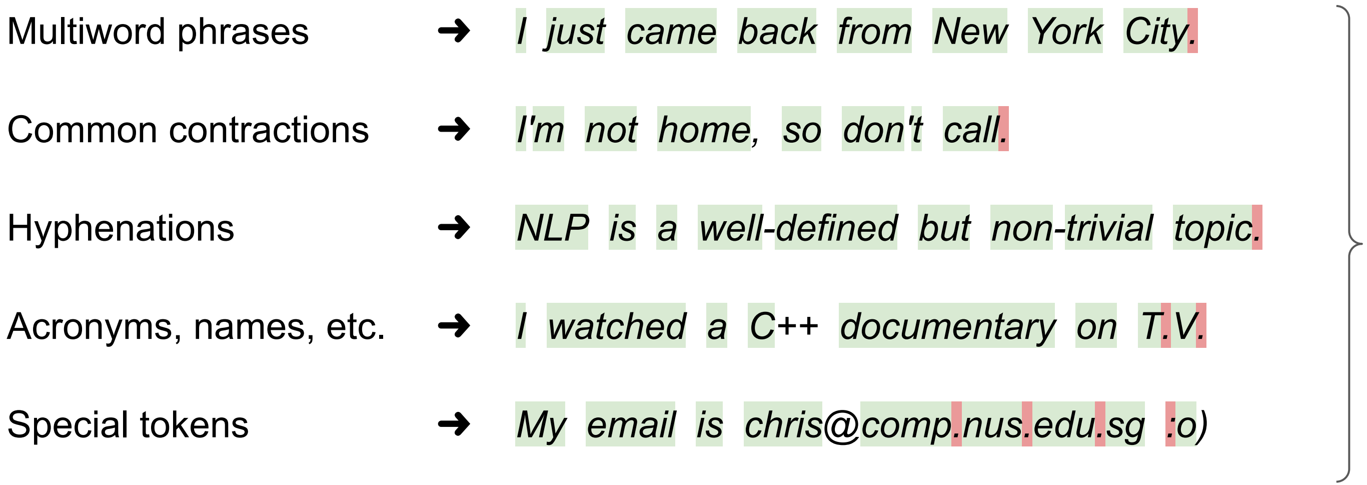
- 2 intuitive approaches (solved using RegEx)
  - Match all words, numbers and punctuation marks ➜ `\w+|\d+|[,.;:]`
  - Match boundaries between "words" and "non-words" ➜ `(?=\W)|(?<=\W)`



`\w+|\d+|[,.;:]` ➜ *NLP is fun, and there is so much to learn in 13 weeks.*

`(?=\W)|(?<=\W)` ➜ *NLP| |is| |fun|,| |and| |there| |is| |so| |much| |to| |learn| |in| |13| |weeks|.|*

Tokenization — It Gets Tricky Pretty Quickly



Multiword phrases ➜ *I just came back from New York City.*

Common contractions ➜ *I'm not home, so don't call.*

Hyphenations ➜ *NLP is a well-defined but non-trivial topic.*

Acronyms, names, etc. ➜ *I watched a C++ documentary on T.V.*

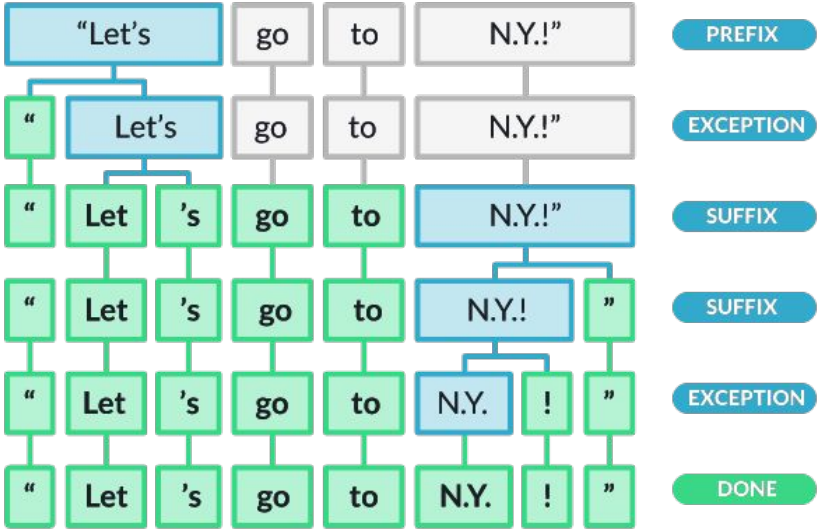
Special tokens ➜ *My email is chris@comp.nus.edu.sg :o)*

RegEx used:

```
\w+|\d+|[,.;:]
```

- Challenges
  - Shape/format of tokens can vary significantly ➜ complex RegEx (or alternative approaches)
  - No officially recognized set of rules for tokenization (different tokenizers might yield different results)

Example: spaCy Tokenizer



(1) Split string on whitespace characters

(2) From left to right, recursively check substrings:

- Does substring match an exception rule?
  (e.g., *"don't"* → *"do"*, *"n't"*, but keep *"U.K."*)
- Can a prefix, suffix or infix be split off?
  (e.g., commas, periods, quotes, hyphens)

Substring checks based on:

- Regular Expressions
- Hand-crafted rules / patterns

Source: https://spacy.io/usage/spacy-101

Tokenization — Language Issues

- French
  - Different uses of apostrophes and hyphens (compared to English)

direct article

l'ensemble

"the whole" / "all"

indicates imperative

donne-moi

"give me!"

➜ 1 token or 2 tokens?

- German
  - Very common: compound nouns

Arbeiterunfallversicherungsgesetz

"worker injury insurance act"

➜ important: **compound splitter**

Tokenization — Language Issues

- Languages without whitespace to separate words

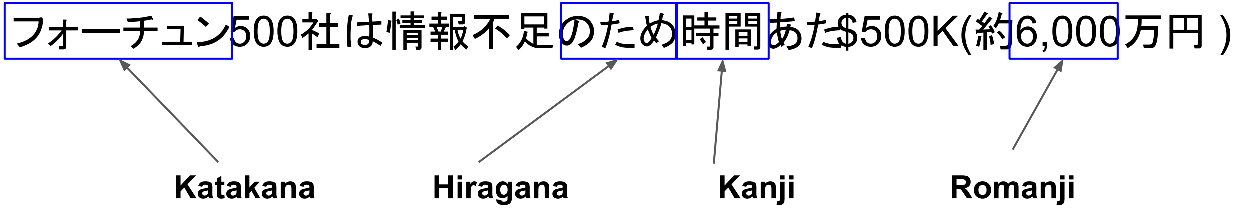
Chinese

莎拉波娃|现在|居住|在|美国|东南部|的|佛罗里达

" Sharapova now lives in US southeastern Florida "

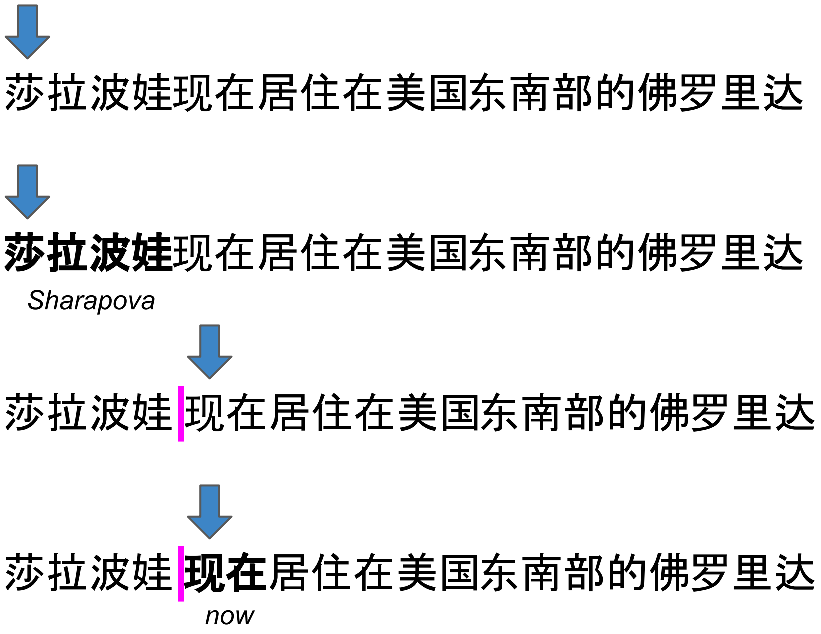
Japanese

- multiple syllabaries
- multiple formats for dates and amounts


フォーチュン500社は情報不足のため時間あた$500K(約6,000万円)
Katakana
Hiragana
Kanji
Romanji

Tokenization — Word Segmentation of Chinese Text

- Baseline algorithm: **Maximum Matching**



莎拉波娃现在居住在美国东南部的佛罗里达

莎拉波娃现在居住在美国东南部的佛罗里达

Sharapova

莎拉波娃|现在居住在美国东南部的佛罗里达

莎拉波娃|**现在**居住在美国东南部的佛罗里达

now

(1) Place a pointer at the beginning of the string

(2) Find longest entry in a dictionary that matches string starting the pointer

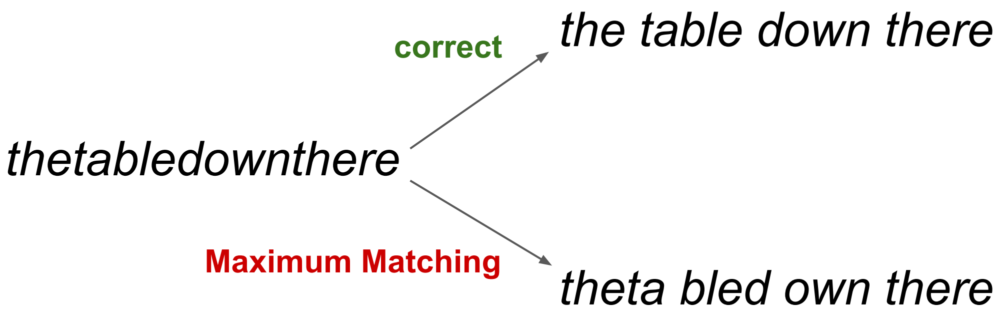
(3) Move the pointer past the identified token in the string

(4) Recurse back to #2, until we process the whole string

Tokenization — Maximum Matching

- Surprisingly good performance on Chinese text
  (even better performance with probabilistic methods or extensions)

- Generally does not work for English text



thetabledownthere

correct

the table down there

Maximum Matching

theta bled own there

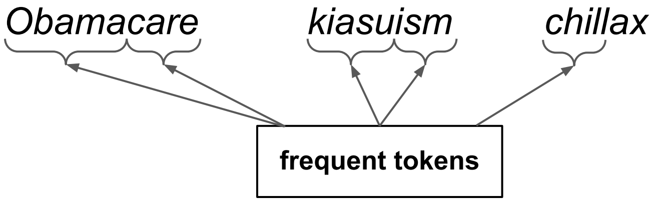
Tokenization — Subword-Based

- Subword-based tokenization
  - So far: *a priori* specification of rules (e.g., RegEx): what constitutes valid tokens?
  - Now: use data to learn how to tokenize

- Why do we want to do this?
  - **Out Of Vocabulary (OOV)** words
    (a word/token an NLP model has not seen before)
  - Very rare words in a corpus

➜ problematic when building statistical models

Examples:



Obamacare *kiasuism* *chillax*

frequent tokens

➜ **Goal:** Split OOV and rare words into (some) known & frequent tokens

Tokenization — Subword-Based

- Different algorithms for subword tokenization
  - Byte-Pair Encoding (BPE), Unigram Language Model Tokenization, WordPiece, etc.

- Different approaches, but similar basic setup

(1) **Token Learner**
Takes raw input (training) corpus and induces a vocabulary
(i.e., set of tokens)

(2) **Token Segmenter**
Takes a raw text and tokenizes it according to the vocabulary

Tokenization — BPE Token Learner

Corpus: *"low low low low low lower lower newest newest newest newest newest newest widest widest widest longer"*

special end-of-word token

```
Initialize vocabulary  (e.g., {'d', 'e', 'g', 'i', 'l', 'n', 'o', 'r', 's', 't', 'w', '_' })
REPEAT
        Find the 2 tokens that are most frequently adjacent to each other (e.g., 'e', 's')
        Add a new merged token 'es' to vocabulary
        Replace every adjacent 'e' 's' in corpus with 'es'
UNTIL k merges have been done
```

parameter of the
algorithm

Tokenization — BPE Token Learner

corpus representation

6	n e w e s t _
5	l o w _
3	w i d e s t _
2	l o w e r _
1	l o n g e r _

vocabulary

d, e, g, i, l, n, o, r, s, t, w, _

merges

most frequent pair: **e** & **s** (9 occurrences)

corpus representation

6	n e w **es** t _
5	l o w _
3	w i d **es** t _
2	l o w e r _
1	l o n g e r _

vocabulary

d, e, g, i, l, n, o, r, s, t, w, _, **es**

merges

(e, s)

most frequent pair: **es** & **t** (9 occurrences)

Tokenization — BPE Token Learner

corpus representation

6	n e w **est** _
5	l o w _
3	w i d **est** _
2	l o w e r _
1	l o n g e r _

vocabulary

d, e, g, i, l, n, o, r, s, t, w, _, es, **est**

merges

(e, s), **(es, t)**

most frequent pair: **est** & **_** (9 occurrences)

corpus representation

6	n e w **est_**
5	l o w _
3	w i d **est_**
2	l o w e r _
1	l o n g e r _

vocabulary

d, e, g, i, l, n, o, r, s, t, w, _, es, est, **est_**

merges

(e, s), (es, t), **(est, _)**

most frequent pair: **l** & **o** (8 occurrences)

Tokenization — BPE Token Learner

corpus representation

6	n e w est_
5	**lo** w _
3	w i d est_
2	**lo** w e r _
1	**lo** n g e r _

vocabulary

d, e, g, i, l, n, o, r, s, t, w, _, es, est, est_, **lo**

merges

(e, s), (es, t), (est, _), **(l, o)**

most frequent pair: **lo** & **w** (7 occurrences)

corpus representation

6	n e w est_
5	**low** _
3	w i d est_
2	**low** e r _
1	lo n g e r _

vocabulary

d, e, g, i, l, n, o, r, s, t, w, _, es, est, est_, lo, **low**

merges

(e, s), (es, t), (est, _), (l, o), **(lo, w)**

most frequent pair: **n** & **e** (6 occurrences)

Tokenization — BPE Token Learner

vocabulary d, e, g, i, l, n, o, r, s, t, w, _, es, est, est_, lo, low, **ne**

merges (e, s), (es, t), (est, _), (l, o), (lo, w), **(n, e)**

vocabulary d, e, g, i, l, n, o, r, s, t, w, _, es, est, est_, lo, low, ne, **new**

merges (e, s), (es, t), (est, _), (l, o), (lo, w), (n, e), **(ne, w)**

vocabulary d, e, g, i, l, n, o, r, s, t, w, _, es, est, est_, lo, low, ne, new, **newest_**

merges (e, s), (es, t), (est, _), (l, o), (lo, w), (n, e), (ne, w), **(new, est_)**

...

Tokenization — BPE Token Segmenter

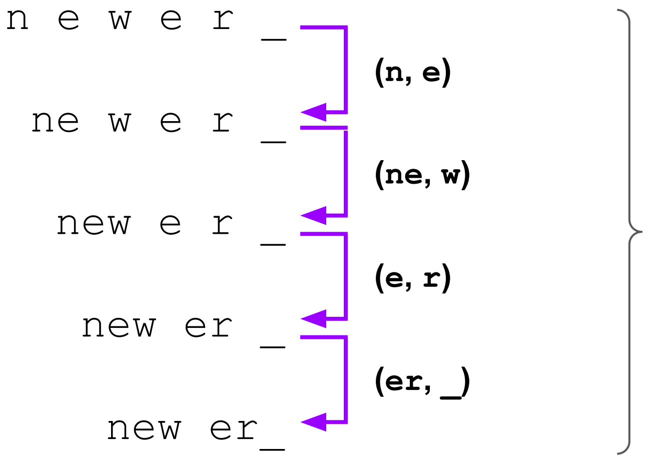
vocabulary d, e, g, i, l, n, o, r, s, t, w, _, es, est, est_, lo, low, ne, new, newest_, low_, er, er_, wi, wid, widest_, lower_, lon, long, longer_

merges (e, s), (es, t), (est, _), (l, o), (lo, w), (n, e), (ne, w), (new, est_), (low, _), (e, r), (er, _), (w, i), (wi, d), (wid, est_), (low, er_), (lo, n), (lon, g), (long, er_)

Tokenize/segment

"newer"

Run each merge in order they have been learned



➜ tokens: *"new"*, *"er_"*

Tokenization — Summary

- Tokenization as a low-level NLP task
  - Challenges: important, non-trivial, language-dependent
  - Particularly tricky for informal language (e.g., social media)

- 3 basic approaches
  - Character-based (trivial to do but often not suitable — individual characters generally carry no semantic meaning)
  - Word-based (a priori specification of rules; language-dependent; problem: OOV/rare words)
  - Subword-based (tokenization learned from data — tokens often turn out to be genuine morphemes!)

- Practical consideration (when using off-the-shell word-based tokenizers)
  - What is my type of text (e.g., formal or informal)? Are there special tokens (e.g., URLs, hashtags)?
  - Try and assess different tokenizers — very, very last resort: write your own tokenizer

NUS | Computing
National University of Singapore

Natural Language Processing: Foundations

Section 2 — Text Preprocessing II (Normalization)

Normalization

- Goal: Convert text into a canonical (standard) form
  - Remove noise / variability / "randomness" from text
  - Affects characters, words, sentences, documents
- Implicit definition of equivalence classes
  - Suitable normalization steps depend on task/application

Alternative to equivalence classes: **asymmetric expansion**

Example: Web Search (utilize case of search terms)

Entered term		**Searched terms**
window	➜	window, windows
windows	➜	Windows, windows, window
Windows	➜	Windows

Raw	Normalized
Germany GERMANY	Germany
USA U.S.A US of A	USA
tonight tonite 2N8	tonight
connect connects connected connecting connection	connect
:) :-) :o)	[EMOTICON+]

Normalization — Case Folding

- When to fold?
  - Common application: Information Retrieval
    (e.g., Web search where most users input in only lowercase anyways)
  - Potential problems: *Trump* vs. *trump*, *MOM* vs. *mom*, *Oracle* vs. *oracle*, etc.
    (potential exception: upper case word in mid sentence?)

- When NOT to fold?
  - NLP tasks where the case of letters or words are important features
  - Examples: Named Entity Recognition, Machine Translation

*They sent **us** a card from the **US** during their vacation.*

Distinction important for NER and MT!

NUS | Computing
National University of Singapore

Natural Language Processing: Foundations

Section 2 — Text Preprocessing III (Stemming & Lemmatization)

Stemming & Lemmatization — Motivating Example

- Two different statements?

"dogs make the best friends" vs. *"a dog makes a good friend"*

➜ Very similar semantics but different syntax

- Common reasons for variations of the same word
  - Singular vs. plural form (mainly of nouns)
  - Different tenses of verbs
  - Comparative/superlative forms of adjectives

➜ Can we normalize words to abstract away such variations?

Normalization — Stemming

- Idea of Stemming
    - Reduce words to their stem
    - Approach: crude chopping of affixes based on rules (➜ language dependent)
    - Different stemmers apply different rules

- Characteristics
    - Pro: fast + no lexicon required
    - Con: stemmed word not necessarily a proper word (i.e., not in dictionary)

Examples
(alternatives reflect results from different stemmers)

Raw	Stemmed
cats	*cat*
running	*run*
phones	*phon(e)*
presumably	*presum*
crying	*cry/cri*
went	*went*
worse	*wors*
best	*best*
mice	*mic(e)*

Normalization — Stemming: Porter Stemmer

- Porter Stemmer — most common stemmer for English text
  - Simple, efficient + very good results in practice

- Series of rewrite rules that run in a cascade
  - Output of each pass is fed is input to the next pass
  - Stemming stops if a pass yields no more changes

	sses → ss	e.g.: *possesses* → *possess*, *classes* → *class*
	tional → tion	e.g., *optional* → *option*, *fictional* → *fiction*
	ies → i	e.g., *cries* → *cri*, *tries* → *tri*
stem must contain vowel →	(*v*)ing → ε	e.g.: *sing* → *sing*, singing → *sing*, *talking* → *talk*
stem must contain >1 chars →	(m>1)ement → ε	e.g., *replacement* → *replac*, *cement* → *cement*

More details: https://tartarus.org/martin/PorterStemmer/

Normalization — Lemmatization

- Idea of Lemmatization
  - Reduce inflections or variant forms to base form
  - Find the correct dictionary headword form
  - Differentiates between word forms: nouns (N), verbs (V), adjectives (A)

Raw	Lemmatized (N)	Lemmatized (V)	Lemmatized (A)
running	*running*	*run*	*running*
phones	*phone*	*phone*	*phones*
went	*went*	*go*	*went*
worse	*worse*	*worse*	*bad*
mice	*mouse*	*mice*	*mice*

Normalization — Lemmatization: Characteristics

- Pros
  - Lemmatized words are proper words (i.e., dictionary words)
  - Can normalize irregular forms (e.g., *went* → *go*, *worst* → *bad*)

- Cons
  - Requires curated lexicons / lookup tables + rules (typically)
  - Requires Part-of-Speech tags for correct results
  - Generally slower than stemming

Normalization — Stemming & Lemmatization

- Back to our motivating examples

Raw:	*"dogs make the best friends"*	*"a dog makes a good friend"*
Stemmed:	*"dog make the best friend"*	*"a dog make a good friend"*
Lemmatized:	*"dog make the good friend"*	*"a dog make a good friend"*

Normalization — Practical Considerations

- Canonical form also affects tokenization, e.g.: Penn Treebank Tokenizer
  - Separate out clitics (e.g., *doesn't* ➜ *does n't*; *John's* ➜ *John 's*)
  - Keep hyphenated words together
  - Separate out all punctuation symbols

- Other common normalization steps
  - Removal of stopwords (e.g., *a*, *an*, *the*, *not*, *and*, *or*, *but*, *to*, *from*, *at*)
  - Removal of non-standard tokens (e.g., URLs, emojis, emoticons)
  - Task-dependent steps (e.g., normalize punctuation marks: *???* ➜ *?*, *?!?!?* ➜ *?*)

Text Preprocessing — Summary

- Goal — Convert raw text to valid input for NLP methods
  - ■ Tokenization: split string into "meaningful units" (i.e., tokens)
  - ■ Normalization: convert text or tokens into a canonical form to reduce complexity (different normalization steps: case folding, stemming / lemmatization, stopword removal, etc.)

- Important — Text preprocessing steps depend on task! — e.g.:
  - ■ Sentiment analysis ➜ Better not remove stopwords such as *"not"* or *"n't"*
  - ■ Named Entity Recognition ➜ Better not case fold (named entities are often capitalized)

➜ Getting text preprocessing right is crucial

- ■ Output directly affects subsequent NLP methods
- ■ Poor preprocessing: *"Garbage in, garbage out"*

NUS | Computing
National University of Singapore

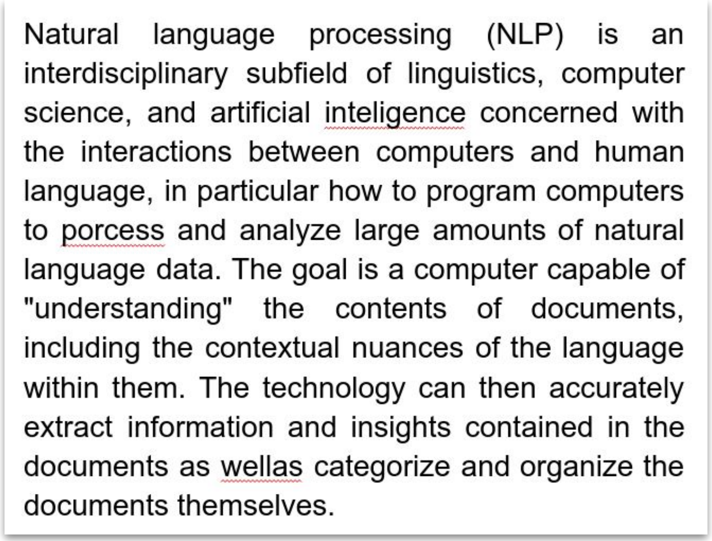
Natural Language Processing: Foundations

Section 2 — Word Error Handling

Motivation

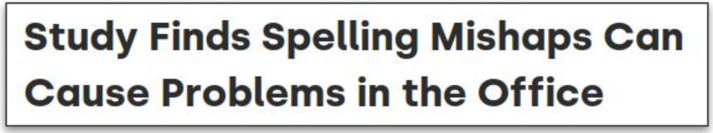
- Despite modern advances in NLP, most text is still written by humans
  - Spoiler: humans are not machines
  - Very common: **spelling errors**
    (and we are also quite bad at catching them)
  - Potentially very negative effects on communication
    (both human-human and human-machine communication)

➜ **Question:** Can automatically identify and maybe even fix spelling errors?

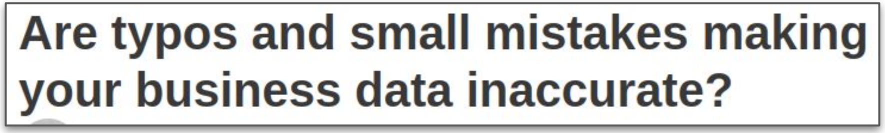


Natural language processing (NLP) is an interdisciplinary subfield of linguistics, computer science, and artificial inteligence concerned with the interactions between computers and human language, in particular how to program computers to porcess and analyze large amounts of natural language data. The goal is a computer capable of "understanding" the contents of documents, including the contextual nuances of the language within them. The technology can then accurately extract information and insights contained in the documents as wellas categorize and organize the documents themselves.

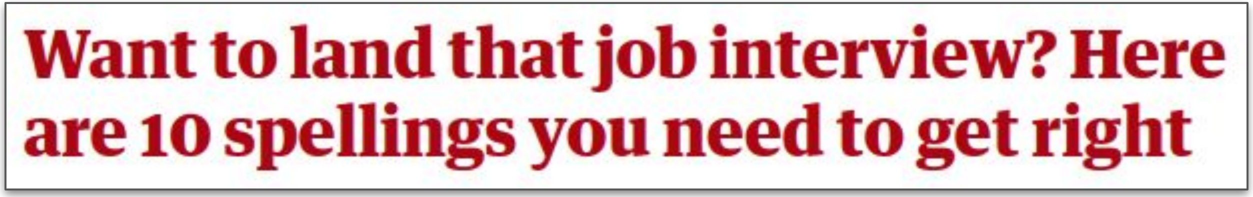
Why Spelling Matters (human–human communication)



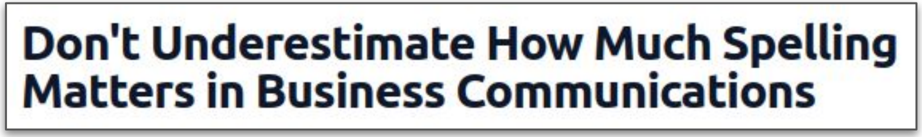
Study Finds Spelling Mishaps Can Cause Problems in the Office



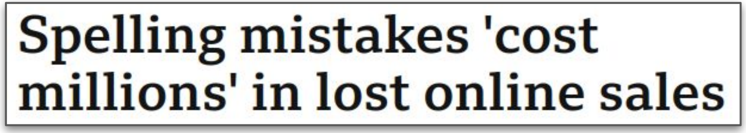
Are typos and small mistakes making your business data inaccurate?



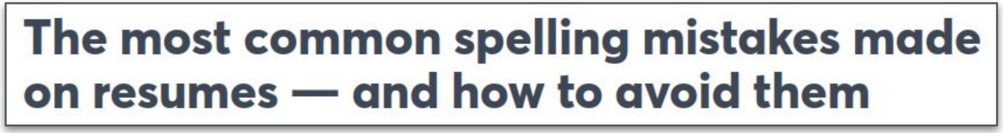
Want to land that job interview? Here are 10 spellings you need to get right



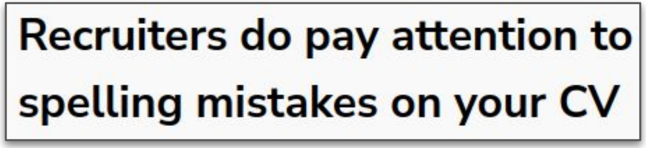
Don't Underestimate How Much Spelling Matters in Business Communications



Spelling mistakes 'cost millions' in lost online sales

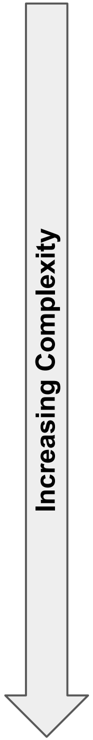


The most common spelling mistakes made on resumes — and how to avoid them



Recruiters do pay attention to spelling mistakes on your CV

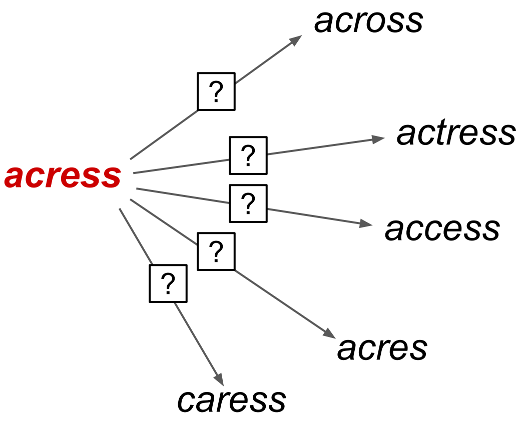
Spelling Errors


Increasing Complexity

1. **Non-word error detections**
   - Basically, word is not found in dictionary
   - Example: detecting *graffe* (misspelling of giraffe)

2. **Isolated-word error correction**
   - Consider word in isolation (i.e., without surrounding words)
   - Example: correcting *graffe* to *giraffe*

3. **Context-sensitive error detection & correction**
   - Consider surrounding words to detect and correct errors
   - Important for "wrong" words that a spelled correctly
   - Examples: *there* vs. *three*, *dessert* vs. *desert*, *son* vs. *song*



Spelling Errors — Common Patterns

- Observation
  - Most misspelled words in typewritten text are **single-error**
  - Damerau (1964): 80%, Peterson (1986): 93-95%

- Single-error misspellings
  - Insertion (e.g., *acress* vs. *acres*)
  - Deletion (e.g., *acress* vs. *actress*)
  - Substitution (e.g., *acress* vs. *access*)
  - Transposition (e.g., *acress* vs. *caress*)

For non-word errors:

➜ Good candidates are orthographically similar

➜ **Minimum Edit Distance**

NUS | Computing
National University of Singapore

Natural Language Processing: Foundations

Section 2 — Word Error Handling I (Minimum Edit Distance)

Minimum Edit Distance (MED)

- Minimum Edit Distance between 2 strings s_1 and s_2
  - Minimum number of allowed edit operations to transform s_1 into s_2
  - Allowed edit operations: **Insertion**, **Deletion**, **Substitution**, Transposition ← Not covered here to keep examples simple

- Example
  - s_1 = *"LANGUAGE"*
  - s_2 = *"SAUSAGE"*

➜ **Alignment** of MED:

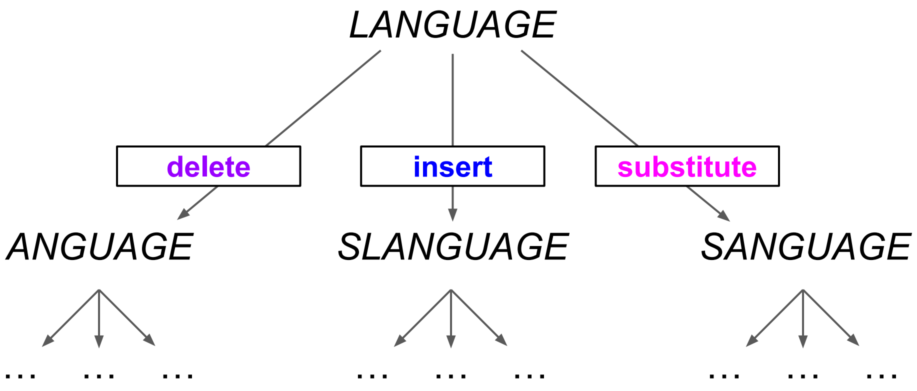
```
L A N G U * A G E
| | | | | | | | |
S A * * U S A G E
```

MED if all operations cost 1 ➜ 4

MED if Substitution costs 2, Insertion 1, Deletion 1 ➜ 5

Minimum Edit Distance — Calculation

- Problem formulation: Find a path (i.e., sequence of edits) from start string to final string
  - **Initial state**: the word being transformed (e.g., *"LANGUAGE"*)
  - **Target state**: the word being transformed into (e.g., *"SAUSAGE"*)
  - **Operators**: **insert**, **delete**, **substitute**
  - **Path cost**: aggregated costs of all edits

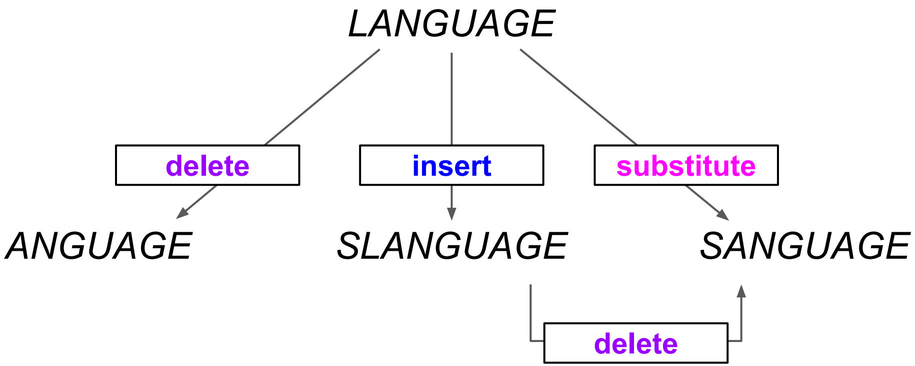


➜ Potentially huge search space

➜ Naive navigation of all path impractical

Minimum Edit Distance — Calculation

- Observations
  - Many distinct paths end up in the same state



➜ No need to keep track of all paths

➜ Only important: shortest path to each revisited state
(shortest in terms of costs, not just number of operations!)

➜ Solve using **Dynamic Programming**

solving problems by combining solutions to subproblems

Minimum Edit Distance — Calculation

- Input: 2 strings
  - Source string X of length n
  - Target string Y of length m

first *i* chars of *X*

first *j* chars of *Y*

- Define $D(i, j)$ as MED between $X[0..i]$ and $Y[0..j]$

➜ MED between X and Y is thus $D(n, m)$

- Bottom-up approach of Dynamic Programming
  - Compute $D(i, j)$ for small i, j (base cases)
  - Compute $D(i, j)$ for larger i, j based on previously computes $D(i, j)$ for smaller i, j

Minimum Edit Distance — Calculation

Assumptions for costs

Insert: **1**

Delete: **1**

Substitute: **2**

➜ Levenshtein MED

- Initialization of bases cases
  - $D(i, 0) = i$ (getting from $X[0..i]$ to empty target string requires i deletions)
  - $D(0, j) = j$ (getting from empty source string to $Y[0..j]$ requires j insertions)

- For $0 < i \leq n$ and $0 < j \leq m$

$$D(i,j) = min \begin{cases} D(i-1,j)+1 & \textbf{Delete} \\ D(i,j-1)+1 & \textbf{Insert} \\ D(i-1,j-1) + \begin{cases} 2, & if\, X[i] \neq Y[j] \\ 0, & if\, X[i] = Y[j] \end{cases} & \textbf{Substitute} \end{cases}$$

Complexity analysis

Space: O(nm)

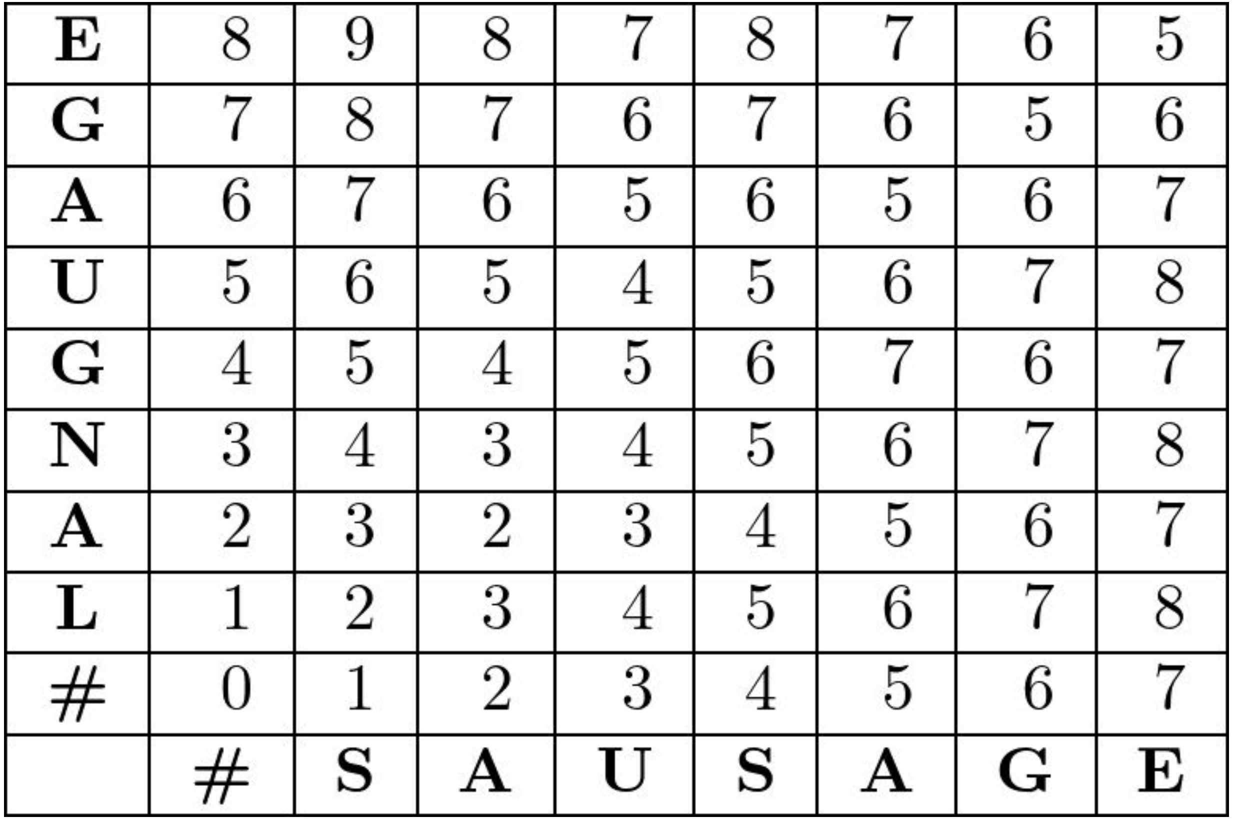
Time: O(nm)

Minimum Edit Distance — Calculation Example

$$D(i,j) = min \begin{cases} D(i-1,j)+1 & \text{Delete} \\ D(i,j-1)+1 & \text{Insert} \\ D(i-1,j-1) + \begin{cases} 2, & if\, X[i] \neq Y[i] \\ 0, & if\, X[i] = Y[i] \end{cases} & \text{Substitute} \end{cases}$$

E	8							
G	7							
A	6							
U	5							
G	4							
N	3							
A	2							
L	1							
#	0	1	2	3	4	5	6	7
	#	S	A	U	S	A	G	E

Minimum Edit Distance — Calculation Example



E	8	9	8	7	8	7	6	5
G	7	8	7	6	7	6	5	6
A	6	7	6	5	6	5	6	7
U	5	6	5	4	5	6	7	8
G	4	5	4	5	6	7	6	7
N	3	4	3	4	5	6	7	8
A	2	3	2	3	4	5	6	7
L	1	2	3	4	5	6	7	8
#	0	1	2	3	4	5	6	7
	#	**S**	**A**	**U**	**S**	**A**	**G**	**E**

Minimum Edit Distance — Backtrace & Alignments

- Current limitation
  - Base algorithms only returns the MED
  - Often important: alignment between strings

```
L A N G U * A G E
| | | | | | | | |
S A * * U S A G E
```

How do we get this?

- Keep track of backtrace
  - Remember from which "direction" we entered a new cell
  - At the end, trace path from upper right corner to read of alignment

Keep set of pointers for each i, j

Small extension to base algorithm:

$$PTR(i,j) = \begin{cases} \texttt{LEFT} & \textbf{Insert} \\ \texttt{DOWN} & \textbf{Delete} \\ \texttt{DIAG} & \textbf{Substitute} \end{cases}$$

Note: Backtraces are generally not unique ➜ different alignments for the same MED possible

Minimum Edit Distance — Backtrace & Alignments

E	8	↙←↓ 9	↓ 8	↓ 7	↙←↓ 8	↓ 7	↓ 6	↙ 5
G	7	↙←↓ 8	↓ 7	↓ 6	↙←↓ 7	↓ 6	↙ 5	← 6
A	6	↙←↓ 7	↙↓ 6	↓ 5	↙←↓ 6	↙ 5	← 6	← 7
U	5	↙←↓ 6	↓ 5	↙ 4	← 5	← 6	←↓ 7	↙←↓ 8
G	4	↙←↓ 5	↓ 4	↙←↓ 5	↙←↓ 6	↙←↓ 7	↙ 6	← 7
N	3	↙←↓ 4	↓ 3	↙←↓ 4	↙←↓ 5	↙←↓ 6	↙←↓ 7	↙←↓ 8
A	2	↙←↓ 3	↙ 2	← 3	← 4	↙← 5	← 6	← 7
L	1	↙←↓ 2	↙←↓ 3	↙←↓ 4	↙←↓ 5	↙←↓ 6	↙←↓ 7	↙←↓ 8
#	0	1	2	3	4	5	6	7
	#	**S**	**A**	**U**	**S**	**A**	**G**	**E**

L A N G U * A G E
| | | | | | | | |
S A * * U S A G E

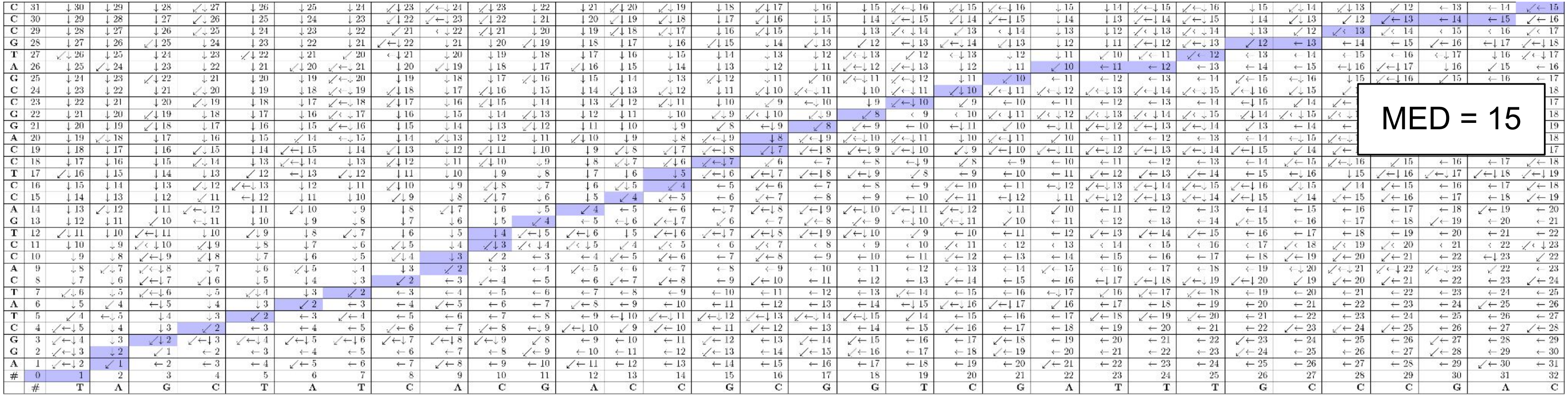
Complexity analysis

Time: O(n+m)

Minimum Edit Distance — More Examples

- Biology: Align 2 sequences of nucleotides

AGGCTATCACCTGACCTCCAGGCCGATGCCC
TAGCTATCACGACCGCGGTCGATTTGCCCGAC



MED = 15

```
* A G G C T A T C A C C T G A C C T C C A G G C C G A * * T G * C C * * C
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
T A * G C T A T C A * C * G A C C * G C * G G T C G A T T T G C C C G A C
```

Minimum Edit Distance — Other Uses in NLP

- Evaluating Machine Translation and speech recognition

  e.g., How similar are 2 translations?

Reference:	*Spokesman*	***confirms***	*	*senior*	***government***	*adviser*	*was*	*shot*	*
Prediction:	*Spokesman*	***said***	***the***	*senior*	*	*adviser*	*was*	*shot*	***dead***

- Named Entity Extraction and Entity Coreference

"We stayed at the	*	*Merchant*	*Court*	*prior to a cruise"*
"The	***Swissotel***	*Merchant*	*Court*	*is a great place to stay in Singapore"*

Referring to the same entity?

Minimum Edit Distance — Extensions

- Weighted Minimum Edit Distance, e.g.:
  - Spelling Correction: some letters are more likely to be mistyped than others
  - Biology: certain forms of deletions or insertions are more likely than others

➜ Generalization of algorithm
  - Application-dependent weights (i.e., costs for edit operations)

Initialization of base cases:

$$\begin{aligned} &D(0,0) = 0 \\ &D(i,0) = D(i-1,0) + del(X[i]), \quad \text{for } 1 < i \leq n \\ &D(0,j) = D(0,j-1) + ins(Y[i]), \quad \text{for } 1 < i \leq m \end{aligned}$$

Recurrence relation:

$$D(i,j) = min \begin{cases} D(i-1,j) & + \, del(X[i]) \\ D(i,j-1) & + \, ins(Y[j]) \\ D(i-1,j-1) & + \, sub(X[i],Y[i]) \end{cases}$$

Minimum Edit Distance — Extensions

- Needleman-Wunsch
  - No penalty for gaps (*) at the beginning or the end of an alignment
  - Good if strings have very different lengths
- Smith-Wasserman
  - Ignore badly aligned regions
  - Find optimal local alignments within substrings
    (Levenshtein finds the best global distance and alignment)

Common application:
Alignment of nucleotides sequences

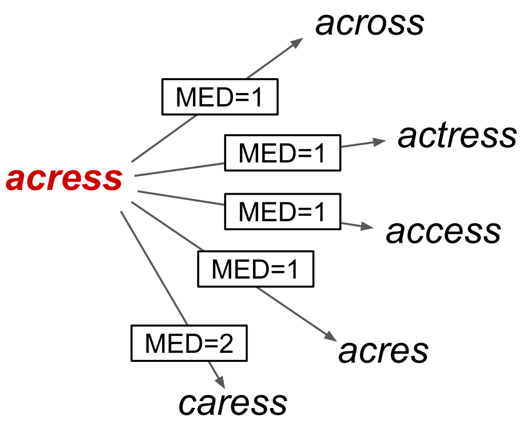
NUS | Computing
National University of Singapore

Natural Language Processing: Foundations

Section 2 — Word Error Handling II (Noisy Channel Model)

Where We are Right Now

- Given a misspelled word, generate suitable candidates for error correction
  - 80% of errors are within minimum edit distance 1
  - Almost all errors within minimum edit distance 2
  - Covers also missing spaces and hyphens
    (e.g., *thisidea* vs. *this idea*; *inlaw* vs. *in-law*)

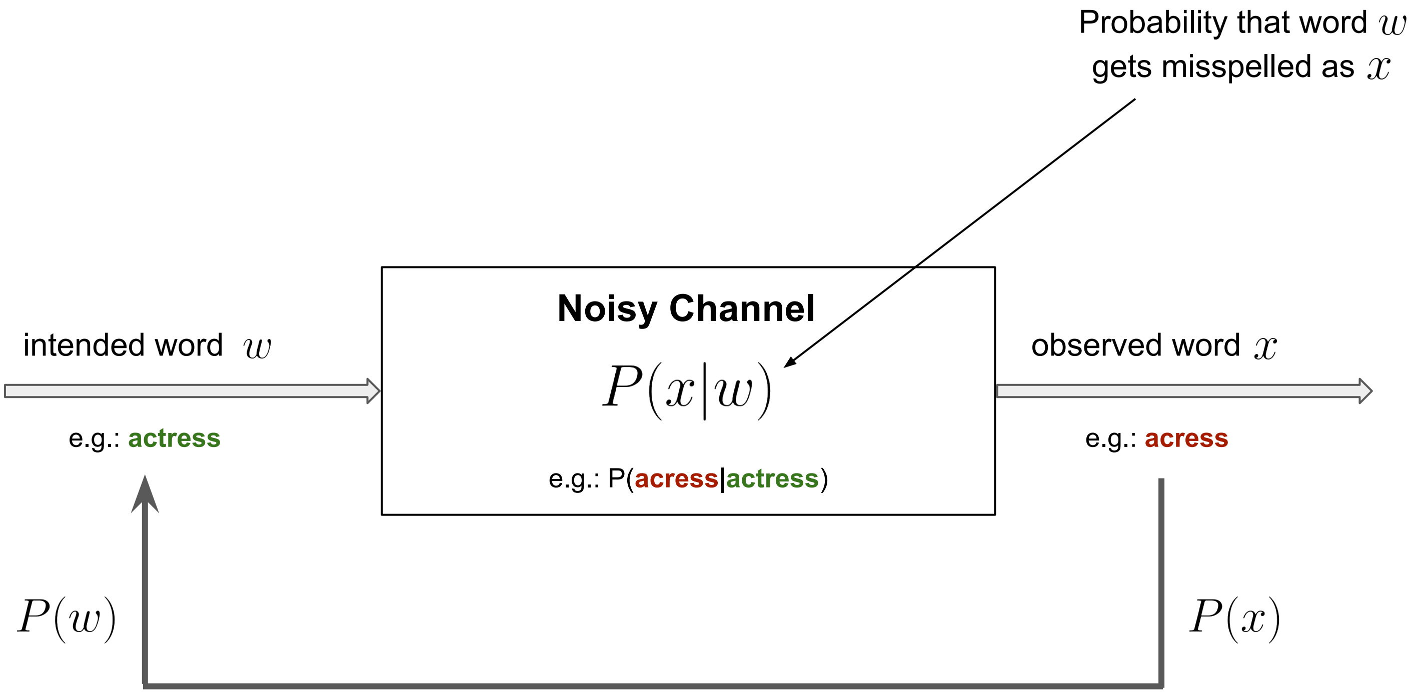


- Still missing: Which is the most likely candidate?
  - Ranking of candidates to show top candidates first
  - Support for automated spelling correction

➜ Noisy Channel Model

Idea: Assign each candidate a probability

Noisy Channel Model — Intuition



Probability that word w gets misspelled as x

intended word w

e.g.: **actress**

Noisy Channel

$P(x|w)$

e.g.: P(**acress**|**actress**)

observed word x

e.g.: **acress**

$P(w)$

$P(x)$

Decoding: Observing error x, can we predict correct word w?

Noisy Channel Model — Bayesian Inferencing

Given an observation x of a misspelled word,

find the correct word w:

$$\widehat{w} = \underset{w \in V}{\operatorname{argmax}}\ P(w|x)$$

$$\widehat{w} = \underset{w \in V}{\operatorname{argmax}}\ \frac{P(x|w)P(w)}{P(x)}$$

$$\widehat{w} = \underset{w \in V}{\operatorname{argmax}}\ P(x|w)P(w)$$

➜ How to calculate $P(x|w)$ and $P(w)$?

Quick refresher: Bayes' Theorem

$$P(A,B) = P(A|B)P(B)$$

$$P(A,B) = P(B|A)P(A)$$

➜ $P(A|B)P(B) = P(B|A)P(A)$

➜ $P(A|B) = \frac{P(B|A)P(A)}{P(B)}$

Noisy Channel Model — Calculating/Estimating $P(w)$

- Approach using Maximum Likelihood Estimate (MLE)
  - Required: Large text corpus with N words
  - Calculate/estimate $P(w)$ with $P(w) = \frac{count(w)}{N}$

- Example
  - 100 MB Wikipedia dump
  - Total of 14.4M+ words

w	*count(w)*	*P(w)*
actress	1,135	0.0000784
cress	1	0.00000…
caress	3	0.00000…
access	1,670	0.0001153
across	1,756	0.0001213
acres	177	0.0000122

Note: The frequencies can widely differ across different corpora (e.g. Wikipedia articles vs. English Literature).

Noisy Channel Model — Calculating/Estimating $P(x|w)$

- In general, $P(x|w)$ almost impossible to predict
  - Predictions depends on arbitrary factors
    (e.g., proficiency of typist, lighting conditions, input device)

- Estimate $P(x|w)$ based on simplifying assumptions (Kernighan et al., 1990)
  - Most misspelled words in typewritten text are single-error
  - Consider only single-error misspellings: **Insertion**, **Deletion**, **Substitution**, **Transposition**

Noisy Channel Model — Calculating/Estimating $P(x|w)$

- Definition of 4 confusion matrices (1 for each single-error type)
  - Each confusion matrix lists the number of times one "thing" was confused with another
  - e.g., for substitution, an entry represents the number of times one letter was incorrectly used

- Underlying definitions for generate confusion matrices

$ins[x, y]$	number of times x was typed as xy
$del[x, y]$	number of times xy was typed as x
$sub[x, y]$	number of times x was substituted for y
$trans[x, y]$	number of times xy was typed as yx
$count[x]$	number of times that x appeared in the training set
$count[x, y]$	number of times that xy appeared in the training set

$$x, y \in \{a, b, c, ..., z\}$$

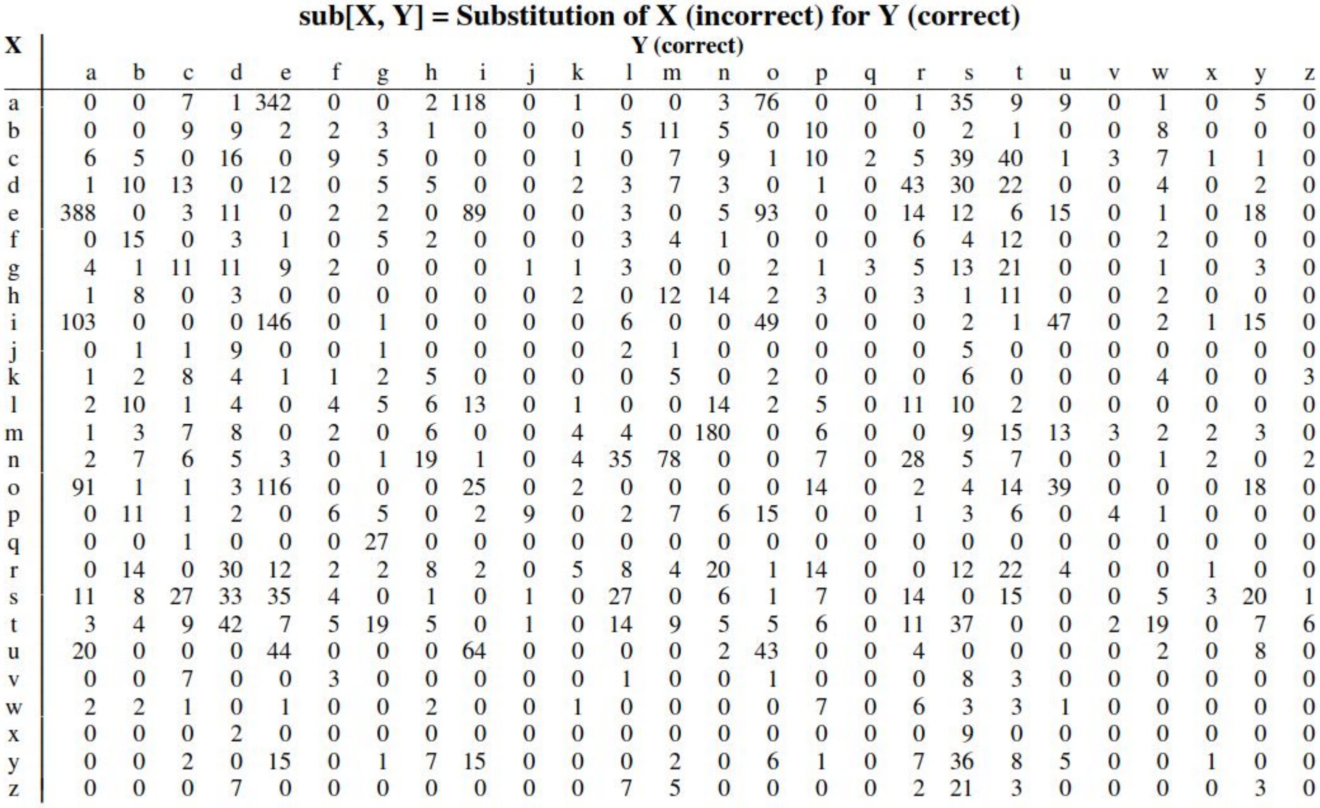
Noisy Channel Model — Calculating/Estimating $P(x|w)$

$$P(x|w) = \begin{cases} \dfrac{ins[w_{i-1},x_i]}{count[w_i]} & \text{, if insertion} \\[2ex] \dfrac{del[w_{i-1},w_i]}{count[w_{i-1,w_i}]} & \text{, if deletion} \\[2ex] \dfrac{sub[x_i,w_i]}{count[w_i]} & \text{, if substitution} \\[2ex] \dfrac{trans[w_i,w_{i+1}]}{count[w_i,w_{i+1}]} & \text{, if transposition} \end{cases}$$

w_i = i-th character in the correct word w

x_i = i-th character in the misspelled word x

Noisy Channel Model — Calculating/Estimating $P(x|w)$



sub[X, Y] = Substitution of X (incorrect) for Y (correct)

X	Y (correct)																									
	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
a	0	0	7	1	342	0	0	2	118	0	1	0	0	3	76	0	0	1	35	9	9	0	1	0	5	0
b	0	0	9	9	2	2	3	1	0	0	0	5	11	5	0	10	0	0	2	1	0	0	8	0	0	0
c	6	5	0	16	0	9	5	0	0	0	1	0	7	9	1	10	2	5	39	40	1	3	7	1	1	0
d	1	10	13	0	12	0	5	5	0	0	2	3	7	3	0	1	0	43	30	22	0	0	4	0	2	0
e	388	0	3	11	0	2	2	0	89	0	0	3	0	5	93	0	0	14	12	6	15	0	1	0	18	0
f	0	15	0	3	1	0	5	2	0	0	0	3	4	1	0	0	0	6	4	12	0	0	2	0	0	0
g	4	1	11	11	9	2	0	0	0	1	1	3	0	0	2	1	3	5	13	21	0	0	1	0	3	0
h	1	8	0	3	0	0	0	0	0	0	2	0	12	14	2	3	0	3	1	11	0	0	2	0	0	0
i	103	0	0	0	146	0	1	0	0	0	0	6	0	0	49	0	0	0	2	1	47	0	2	1	15	0
j	0	1	1	9	0	0	1	0	0	0	0	2	1	0	0	0	0	0	5	0	0	0	0	0	0	0
k	1	2	8	4	1	1	2	5	0	0	0	0	5	0	2	0	0	0	6	0	0	0	4	0	0	3
l	2	10	1	4	0	4	5	6	13	0	1	0	0	14	2	5	0	11	10	2	0	0	0	0	0	0
m	1	3	7	8	0	2	0	6	0	0	4	4	0	180	0	6	0	0	9	15	13	3	2	2	3	0
n	2	7	6	5	3	0	1	19	1	0	4	35	78	0	0	7	0	28	5	7	0	0	1	2	0	2
o	91	1	1	3	116	0	0	0	25	0	2	0	0	0	0	14	0	2	4	14	39	0	0	0	18	0
p	0	11	1	2	0	6	5	0	2	9	0	2	7	6	15	0	0	1	3	6	0	4	1	0	0	0
q	0	0	1	0	0	0	27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
r	0	14	0	30	12	2	2	8	2	0	5	8	4	20	1	14	0	0	12	22	4	0	0	1	0	0
s	11	8	27	33	35	4	0	1	0	1	0	27	0	6	1	7	0	14	0	15	0	0	5	3	20	1
t	3	4	9	42	7	5	19	5	0	1	0	14	9	5	5	6	0	11	37	0	0	2	19	0	7	6
u	20	0	0	0	44	0	0	0	64	0	0	0	0	2	43	0	0	4	0	0	0	0	2	0	8	0
v	0	0	7	0	0	3	0	0	0	0	0	1	0	0	1	0	0	0	8	3	0	0	0	0	0	0
w	2	2	1	0	1	0	0	2	0	0	1	0	0	0	0	7	0	6	3	3	1	0	0	0	0	0
x	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9	0	0	0	0	0	0	0
y	0	0	2	0	15	0	1	7	15	0	0	0	2	0	6	1	0	7	36	8	5	0	0	1	0	0
z	0	0	0	7	0	0	0	0	0	0	0	7	5	0	0	0	0	2	21	3	0	0	0	0	3	0

Source: A Spelling Correction Program Based on a Noisy Channel Model (Kernighan et al., 1990)

Noisy Channel Model — Example

- Noisy channel probabilities for *"acress"*

Candidate Correction	Correct Letter	Error Letter	x|w	P(x|w)	P(w)	10^9*P(x|w)P(w)
actress	t		c|ct	.000117	.0000784	**9.2**
cress		a	a|#	.00000144	.00000…	.000…
caress	ca	ac	ac|ca	.00000164	.00000…	.000…
access	c	r	r|c	.00000021	.0001153	0.024
across	o	e	e|o	.0000093	.0001213	1.12
acres		s	es|e	.0000321	.0000122	0.39
acres		s	ss|s	.0000342	.0000122	0.41

➜ Choice of candidate for correction: *actress*

Noisy Channel Model — Discussion

- Basic limitation: No consideration of additional context
  - Model only applicable for non-word errors
  - Basic model will always suggest "actress" to correct "acress"

"I saw him from ***acress*** *the street."*

"across" here the better candidate

➜ **Language Models** (next section!)

Summary

- RegEx — fundamental and useful tool

- Text Preprocessing — getting your data ready for analysis
  - Tokenization
  - Stemming / Lemmatization
  - Normalization

  typical very task-dependent!

- Error Handling (so far)
  - Focus on single-error misspellings
  - Focus on isolated-word error correction

  already very non-trivial!