Social Media & Text Analysis

lecture 4 - natural language processing (part 2): tokenization and normalization

Follow @cocoweixu

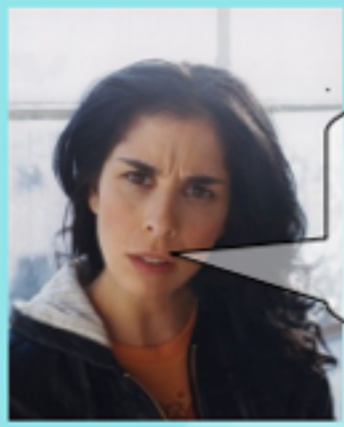
Instructor: Wei Xu

Website: socialmedia-class.org

BAD LANGUAGE!

...on the INTERNET!!

Jacob EISENSTEIN
GEORGIA Institute of TECHnology



Boom! Ya ur website suxx bro

...dats why pluto is pluto it can neva be a star

michelle obama great. job. and. whit all my. respect she. look. great. congrats. to. her.

I now h v an iphone

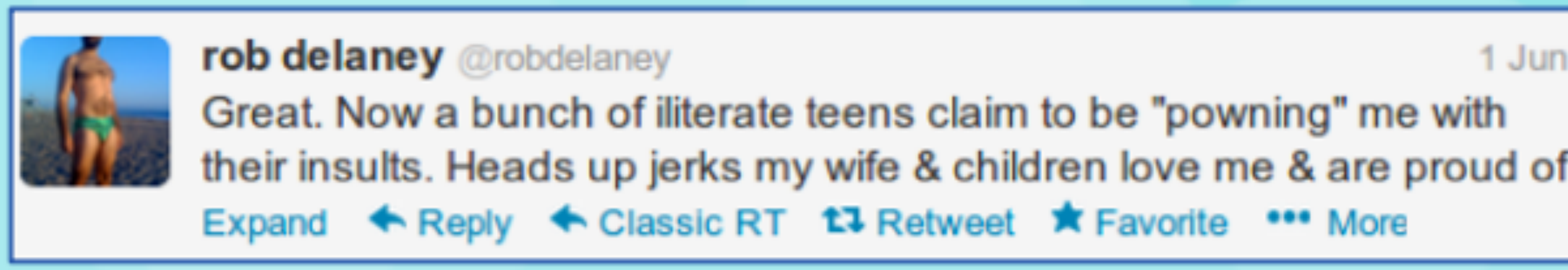
What can we do about it?

*Why don't they just write **NORMALLY**??*

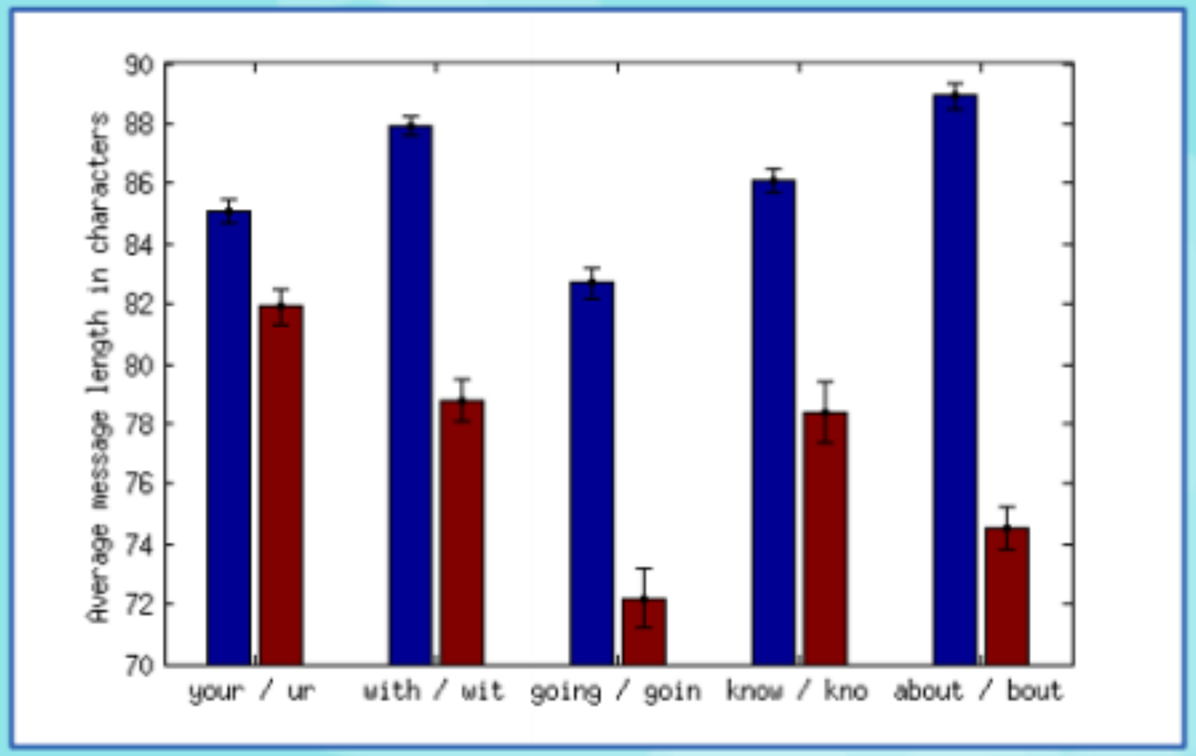
*Can our software ever **ADAPT**???*

How does language go bad?

Illiteracy? No.
(Tagliamonte and Denis 2008; Drouin and Davis 2009)



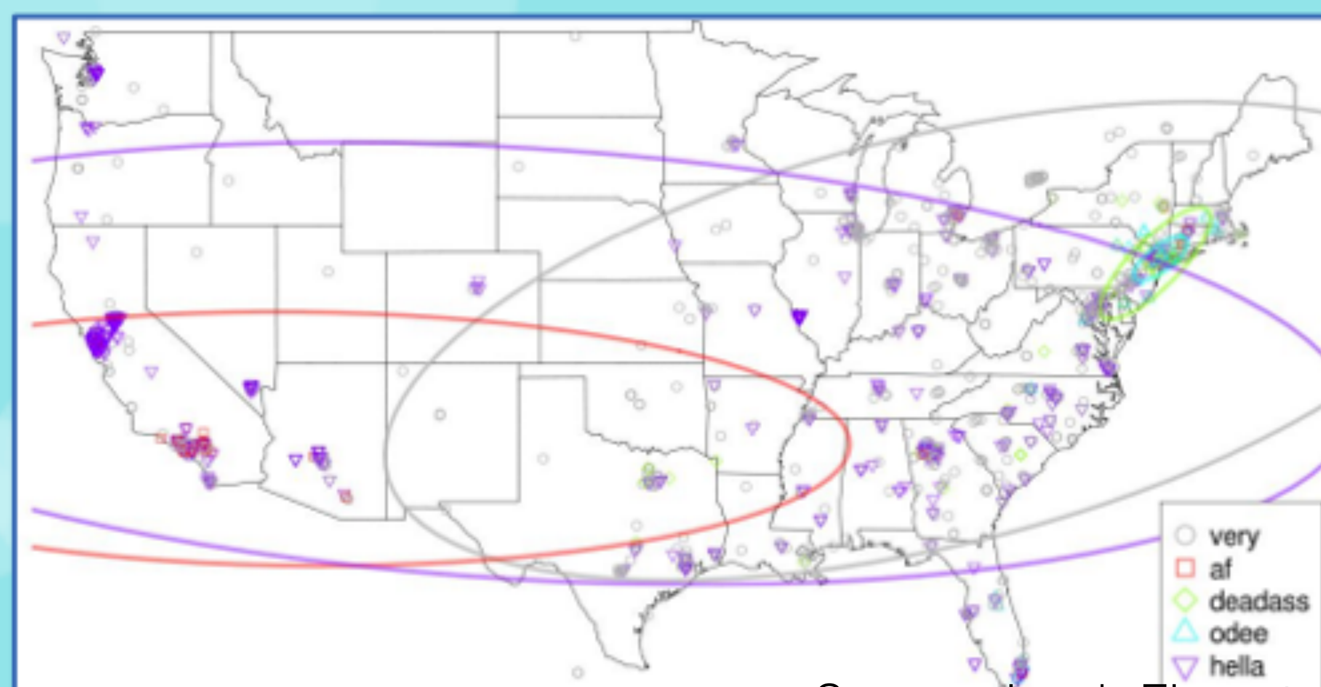
Length limits? (probably not)



Hardware input constraints?
(Gouws et al 2011)

Social variables

- Non-standard language does *identity work*, signaling authenticity, solidarity, etc.
- Social variation is usually inhibited in written language, but social media is less regulated than other written genres.



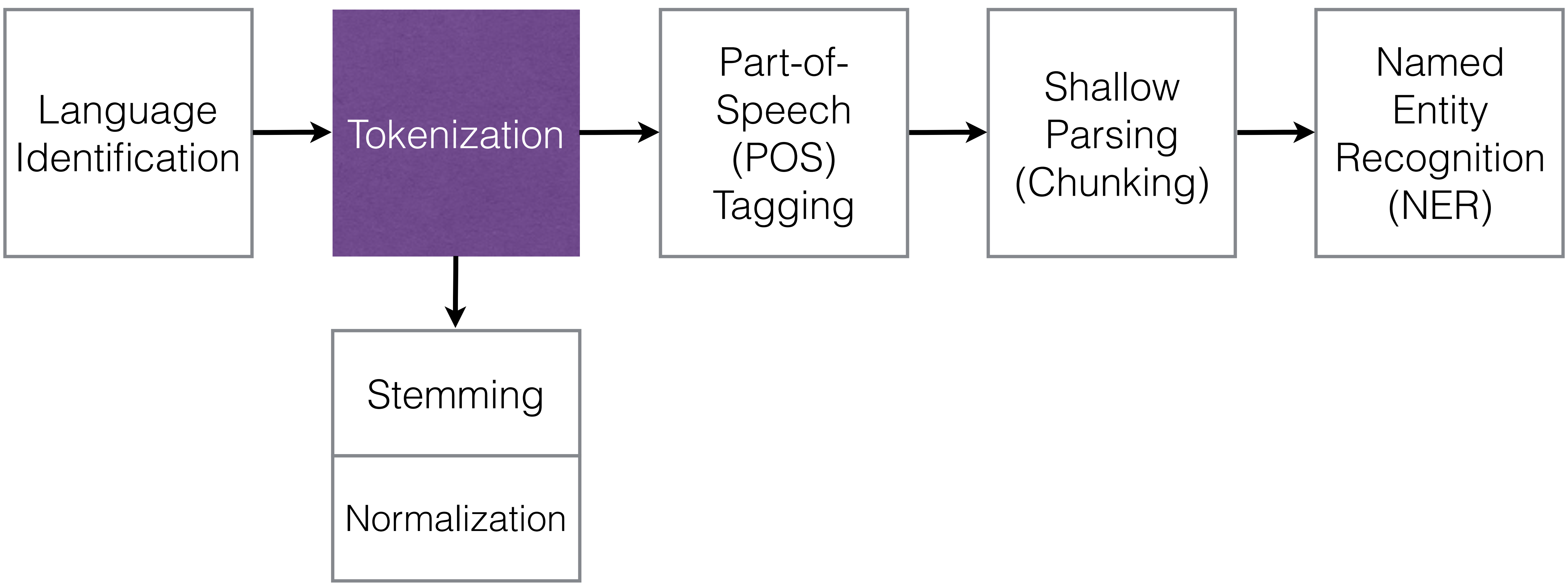
Source: Jacob Eisenstein

Why is Social Media Text “Bad”?

- Lack of literacy? no [Drouin and Davis, 2009]
- Length restrictions? not primarily [Eisenstein, 2013]
- Text input method? to some degree, yes [Gouws et al., 2011]
- Pragmatics (mimicking prosodic effects etc. in speech)? yeeees [Eisenstein, 2013]
- Social variables/markers of social identity? blood oath! [Eisenstein, 2013]

Source: Jacob Eisenstein & Tim Baldwin

NLP Pipeline



Tokenization

- breaks up the string into words and punctuation
- need to handle:
  - abbreviations (“jr.”), number (“5,000”) …

```
seas479:training weixu$ ./penn-treebank-tokenizer.perl
Tokenizer v3
Language: en


Ms. Hilton last year called Mr. Rothschild “the love of my life.”
Ms. Hilton last year called Mr. Rothschild “ the love of my life . ”
```

input

output

Tokenization

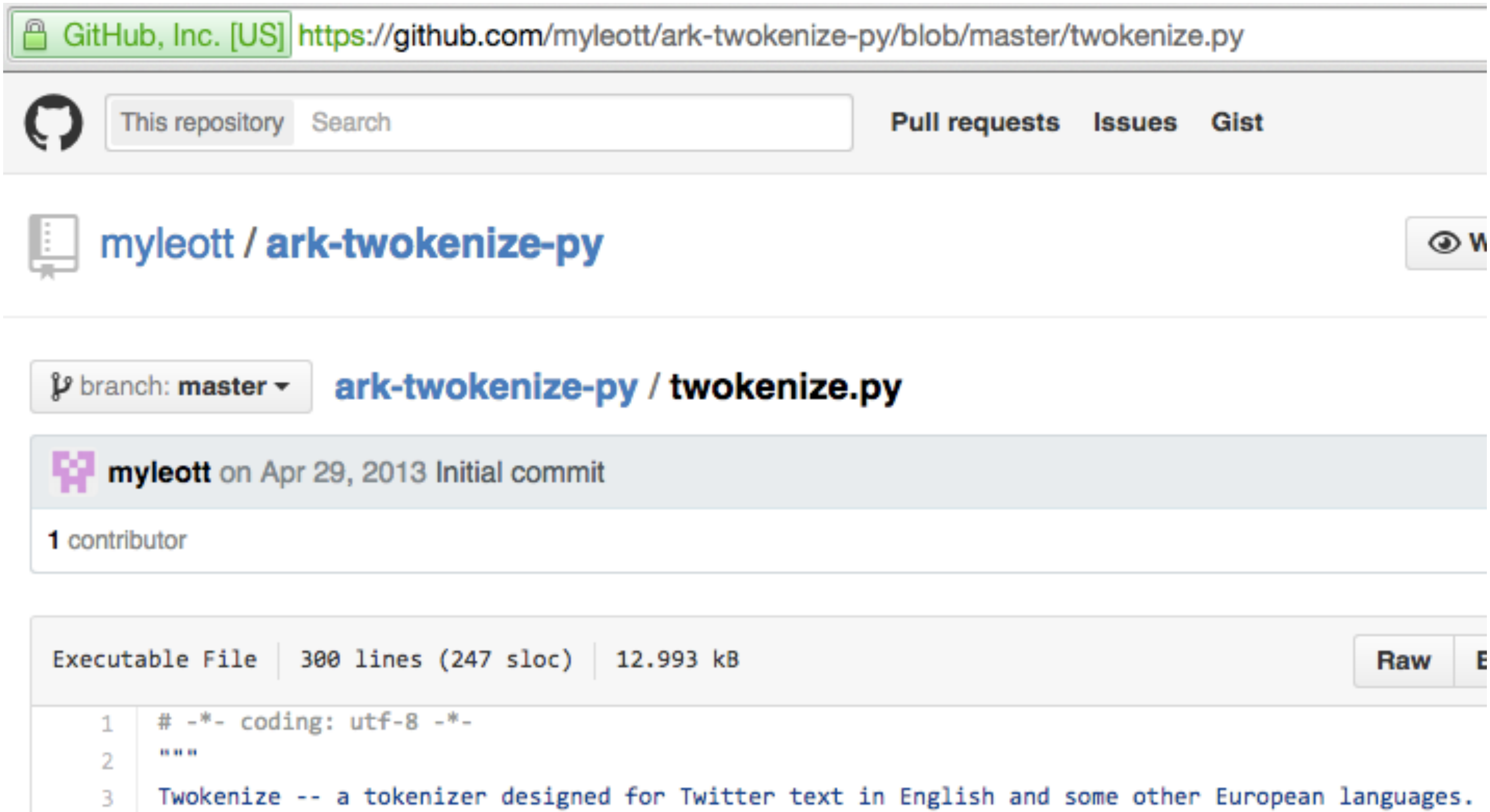
- for Twitter, additionally need to handle:
  - emoticons, urls, #hashtags, @mentions …

```
>>> import twokenize
>>> input = "Clowns are pretty gross tho O.o (I'm afraid of clow
ns :p) ask.fm/a/cc301167"
>>> twokenize.tokenizeRawTweetText(input)
['Clowns', 'are', 'pretty', 'gross', 'tho', 'O.o', '(', "I'm", '
afraid', 'of', 'clowns', ':p', ')', 'ask.fm/a/cc301167']
```

input

output

Tool: twokenize.py



Tool: twokenize.py

```
Twokenize -- a tokenizer designed for Twitter text in English and some other European languages.
This tokenizer code has gone through a long history:

(1) Brendan O'Connor wrote original version in Python, http://github.com/brendano/tweetmotif
       TweetMotif: Exploratory Search and Topic Summarization for Twitter.
       Brendan O'Connor, Michel Krieger, and David Ahn.
       ICWSM-2010 (demo track), http://brenocon.com/oconnor_krieger_ahn.icwsm2010.tweetmotif.pdf
(2a) Kevin Gimpel and Daniel Mills modified it for POS tagging for the CMU ARK Twitter POS Tagger
(2b) Jason Baldridge and David Snyder ported it to Scala
(3) Brendan bugfixed the Scala port and merged with POS-specific changes
    for the CMU ARK Twitter POS Tagger
(4) Tobi Owoputi ported it back to Java and added many improvements (2012-06)

Current home is http://github.com/brendano/ark-tweet-nlp and http://www.ark.cs.cmu.edu/TweetNLP
```

Tokenization

- main techniques:
  - hand-crafted rules as regular expressions

Regular Expression

- a pattern matching language
- invented by American Mathematician Stephen Kleene in the 1950s
- used for search, find, replace, validation … (very frequently used when dealing with strings)
- supported by most programming languages
- easy to learn, but hard to master

Regular Expression

```
147    Hashtag = "#[a-zA-Z0-9_]+"
```

- [] indicates a set of characters:
  - [amk] will match ‘a’, ‘m’, or ‘k’
  - [a-z] will match any lowercase letter (‘abcdefghijklmnopqrstuvwxyz’)
  - [a-zA-Z0-9_] will match any letter or digit or ‘_’
- + matches 1 or more repetitions of preceding RE

Regular Expression

```
Hashtag = "#[a-zA-Z0-9_]+"
```

- will match strings that:
    - start with a ‘#’
    - follow with one or more letters/digits/‘_’

Regular Expression

```
147    Hashtag = "#[a-zA-Z0-9_]+"
```

```
>>> import re
>>> Hashtag = "#[a-zA-Z0-9_]+"
>>> hashtagpattern = re.compile(Hashtag)
>>> hashtagpattern.findall("So that's what #StarWars")
['#StarWars']
```

Regular Expression

```
Hearts = "(?:<+/?3+)+"
```

- ‘+’ matches 1 or more repetitions of the preceding RE
  - ‘<+’ matches ‘<’, ‘<<’, ‘<<<’ …
  - ‘3+’ matches ‘3’, ‘33’, ‘333’ …
- ‘?’ matches 0 or 1 repetitions of the preceding RE
  - ‘/?’ matches ‘/’ or nothing (so handles ‘</3’)
- (?: …) is a non-capturing version of (…)
- (…) matches whatever RE is inside the parentheses

Regular Expression

```
Hearts = "(?:<+/?3+)+"
```

- will match strings that:
  - start with one or more ‘<‘
  - then maybe a ‘/’
  - then one or more ‘3’
  - and maybe repetitions of the above

Regular Expression

```
Hearts = "(?:<+/?3+)+"
```

```
>>> import re
>>> Hearts = "(?:<+/?3+)+"
>>> heartspattern = re.compile(Hearts)
>>> heartspattern.findall("I <3 u <3<333333")
['<3', '<3<333333']
>>> heartspattern.findall("sooo sad </3")
['</3']
```

Regular Expression

- learn more (https://docs.python.org/2/library/re.html)

Python » 2.7.10 Documentation » The Python Standard Library » 7. String Services » previous | next | modules | index

Table Of Contents

7.2. `re` — Regular expression operations

This module provides regular expression matching operations similar to those found in Perl. Both patterns and strings to be searched can be Unicode strings as well as 8-bit strings.

Regular expressions use the backslash character (`'\'`) to indicate special forms or to allow special characters to be used without invoking their special meaning. This collides with Python's usage of the same character for the same purpose in string literals; for example, to match a literal backslash, one might have to write `'\\\\'` as the pattern string, because the regular expression must be `\\`, and each backslash must be expressed as `\\` inside a regular Python string literal.

The solution is to use Python's raw string notation for regular expression patterns; backslashes are not handled in any special way in a string literal prefixed with `'r'`. So `r"\n"` is a two-character string containing `'\'` and `'n'`, while `"\n"` is a one-character string containing a newline. Usually patterns will be expressed in Python code using this raw string notation.

It is important to note that most regular expression operations are available as module-level functions and `RegexObject` methods. The functions are shortcuts that don't require you to compile a regex object first, but miss some fine-tuning parameters.

7.2.1. Regular Expression Syntax

Tokenization

- for Twitter, additionally need to handle:
  - emoticons, urls, #hashtags, @mentions …

```
>>> import twokenize
>>> input = "Clowns are pretty gross tho O.o (I'm afraid of clow
ns :p) ask.fm/a/cc301167"
>>> twokenize.tokenizeRawTweetText(input)
['Clowns', 'are', 'pretty', 'gross', 'tho', 'O.o', '(', "I'm", '
afraid', 'of', 'clowns', ':p', ')', 'ask.fm/a/cc301167']
```

input

output

Tokenization

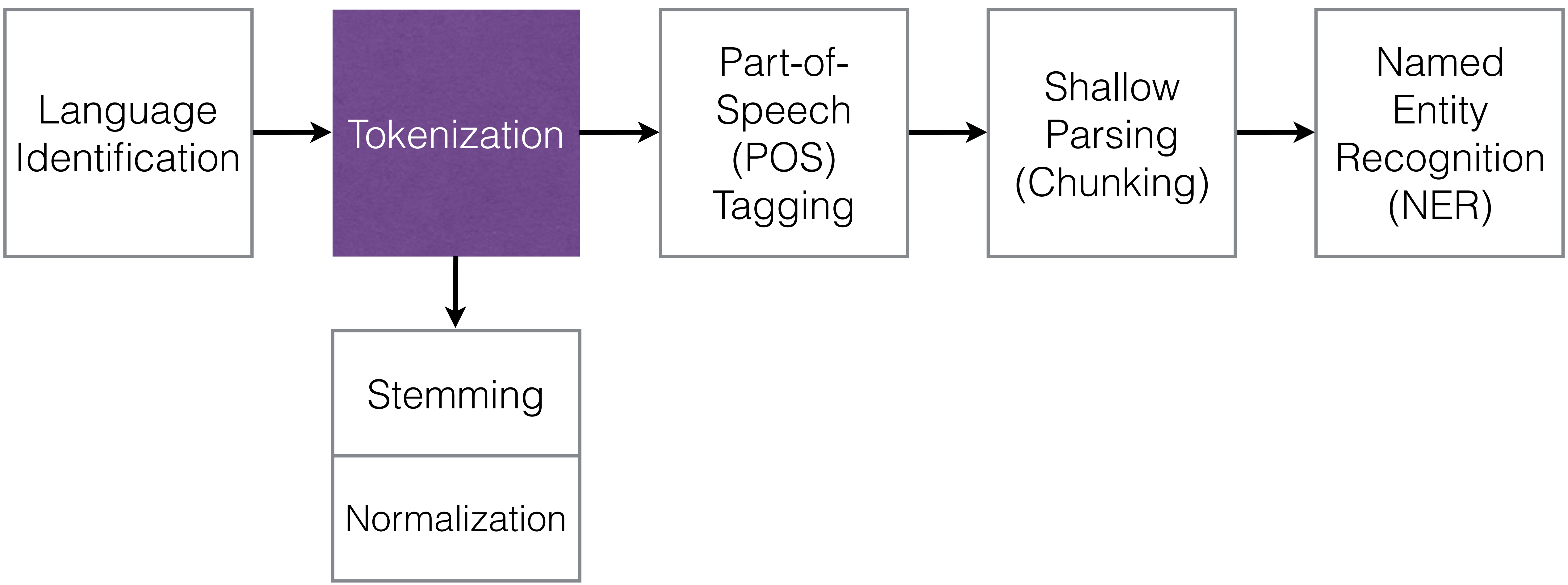
- language dependent

下雨天留客天留我不留	Unpunctuated Chinese sentence
下雨、天留客。天留、我不留！	*It is raining, the god would like the guest to stay. Although the god wants you to stay, I do not!*
下雨天、留客天。留我不？留！	*The rainy day, the staying day. Would you like me to stay? Sure!*

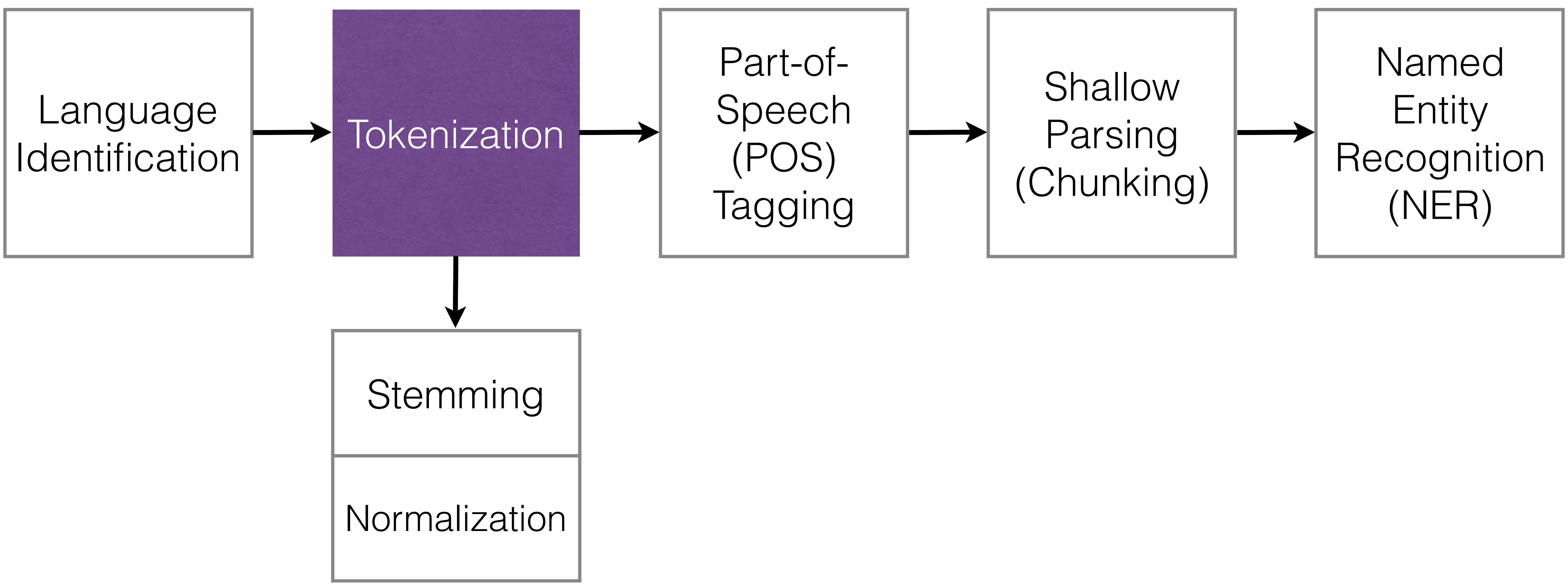
我喜欢新西兰花	Unsegmented Chinese sentence
我 喜欢 新西兰 花	*I like New Zealand flowers*
我 喜欢 新 西兰花	*I like fresh broccoli*

Source: http://what-when-how.com

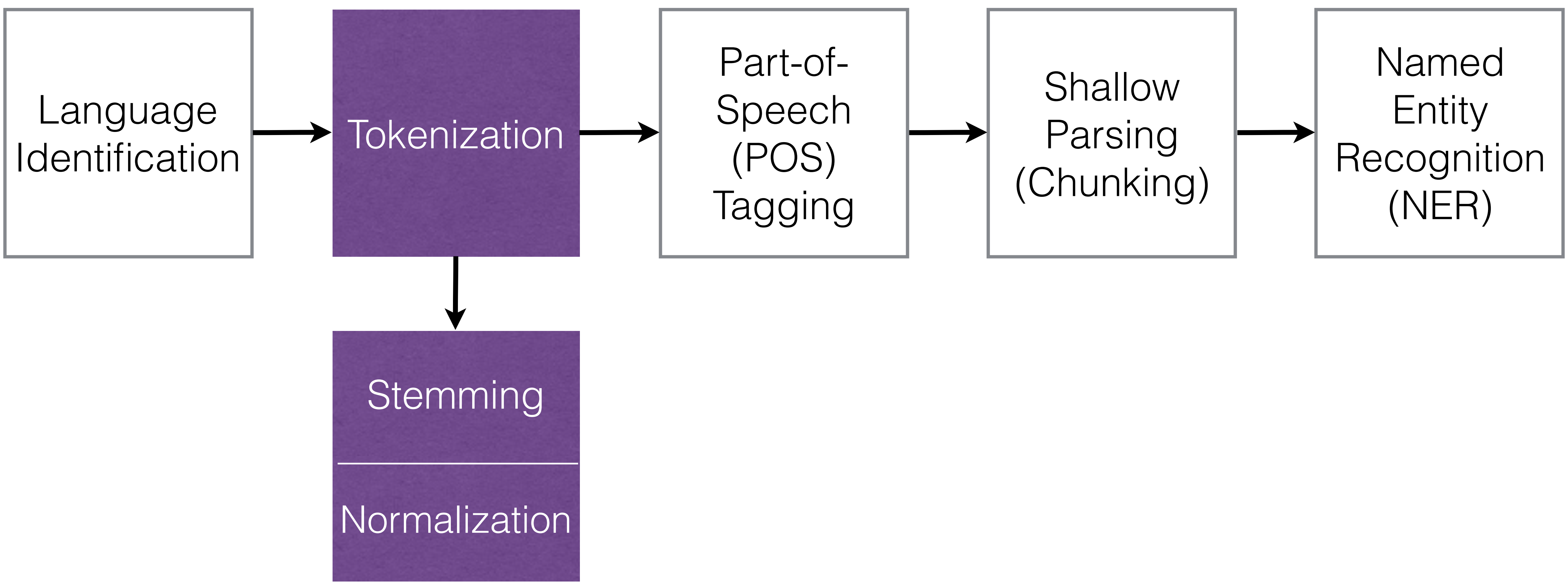
NLP Pipeline



NLP Pipeline



NLP Pipeline



Stemming

- reduce inflected words to their word stem, base or root form (not necessarily the morphological root)

- studied since the 1960s

```
>>> from nltk.stem.porter import PorterStemmer
>>> porter_stemmer = PorterStemmer()
>>> porter_stemmer.stem('maximum')
'maximum'
>>> porter_stemmer.stem('presumably')
'presum'
>>> porter_stemmer.stem('multiply')
'multipli'
```

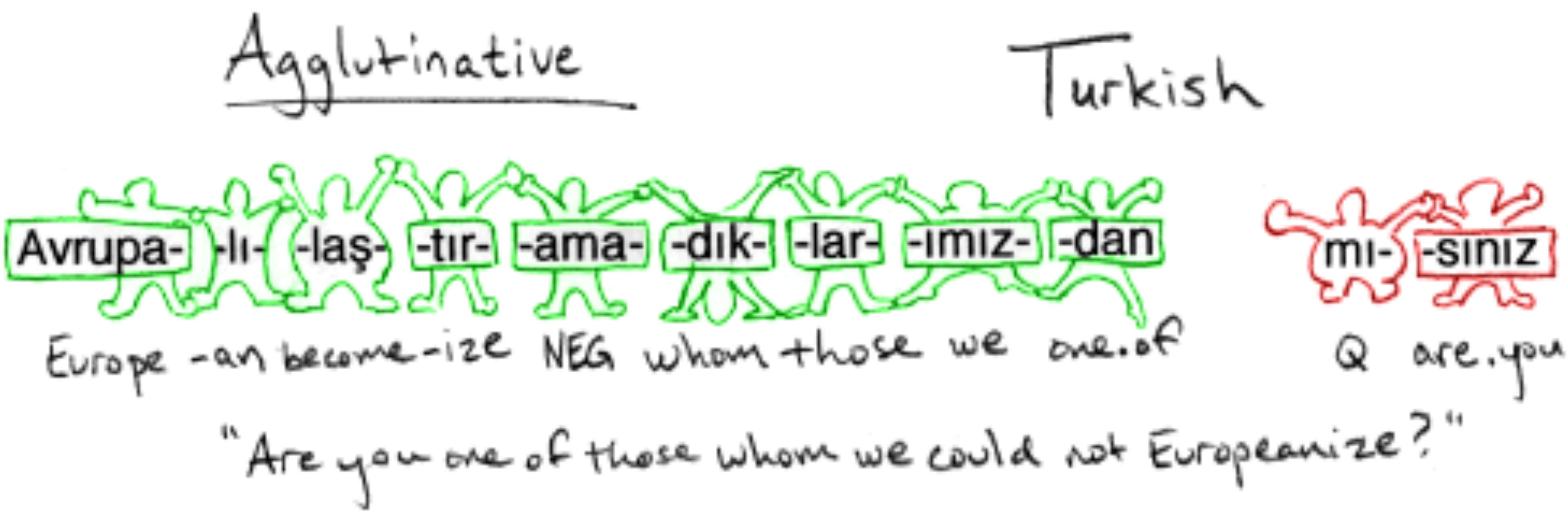
Stemming

- different steamers: Porter, Snowball, Lancaster …
- WordNet's built-in lemmatized (dictionary-based)

```
>>> from nltk.stem import WordNetLemmatizer
>>> wordnet_lemmatizer = WordNetLemmatizer()
>>> wordnet_lemmatizer.lemmatize('leaves', pos='n')
'leaf'
>>> wordnet_lemmatizer.lemmatize('leaves', pos='v')
'leave'
```

Stemming

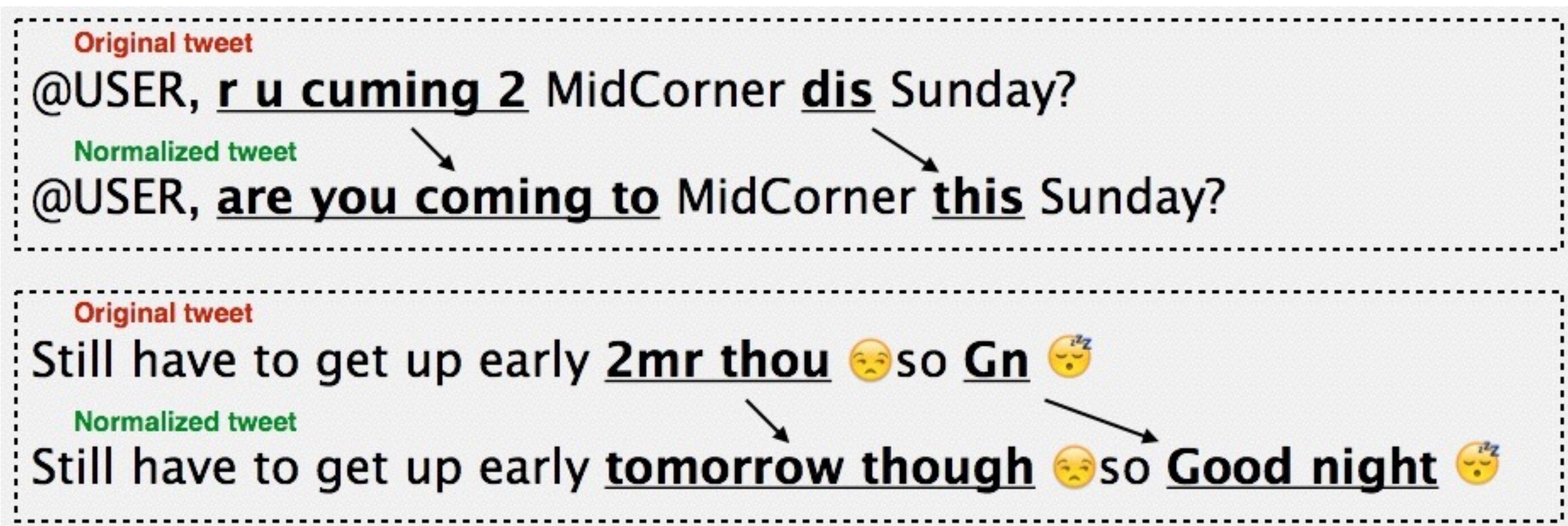
- language dependent



Source: All Things Linguistic

Text Normalization

- convert non-standard words to standard



Source: Baldwin, de Marneffe, Han, Kim, Ritter, Xu
Shared Tasks of the 2015 Workshop on Noisy User-generated Text:
Twitter Lexical Normalization and Named Entity Recognition

Text Normalization

- types of non-standard words in 449 English tweets:

Category	Ratio	Example
letter&numer	2.36%	b4 → before
letter	72.44%	shuld → should
number substitution	2.76%	4 → for
slang	12.20	lol → laugh out loud
other	10.24%	sucha → such a

most non-standard words are morphophonemic “errors”

Source: Bo Han and Timothy Baldwin
“Lexical normalisation of short text messages: Makn sens a #twitter” ACL 2011

A Normalization Lexicon

- automatically derived from Twitter data + dictionary

```
costumess   costumes
nywhere anywhere
sandwhich   sandwich
aleksander  alexander
juns    jun
showi   showing
washinq washing
jscript script
fundin  funding
itxted  fitted
cheeeap cheap
fawesome    awesome
untalented  talented
```

<u>Performance</u>

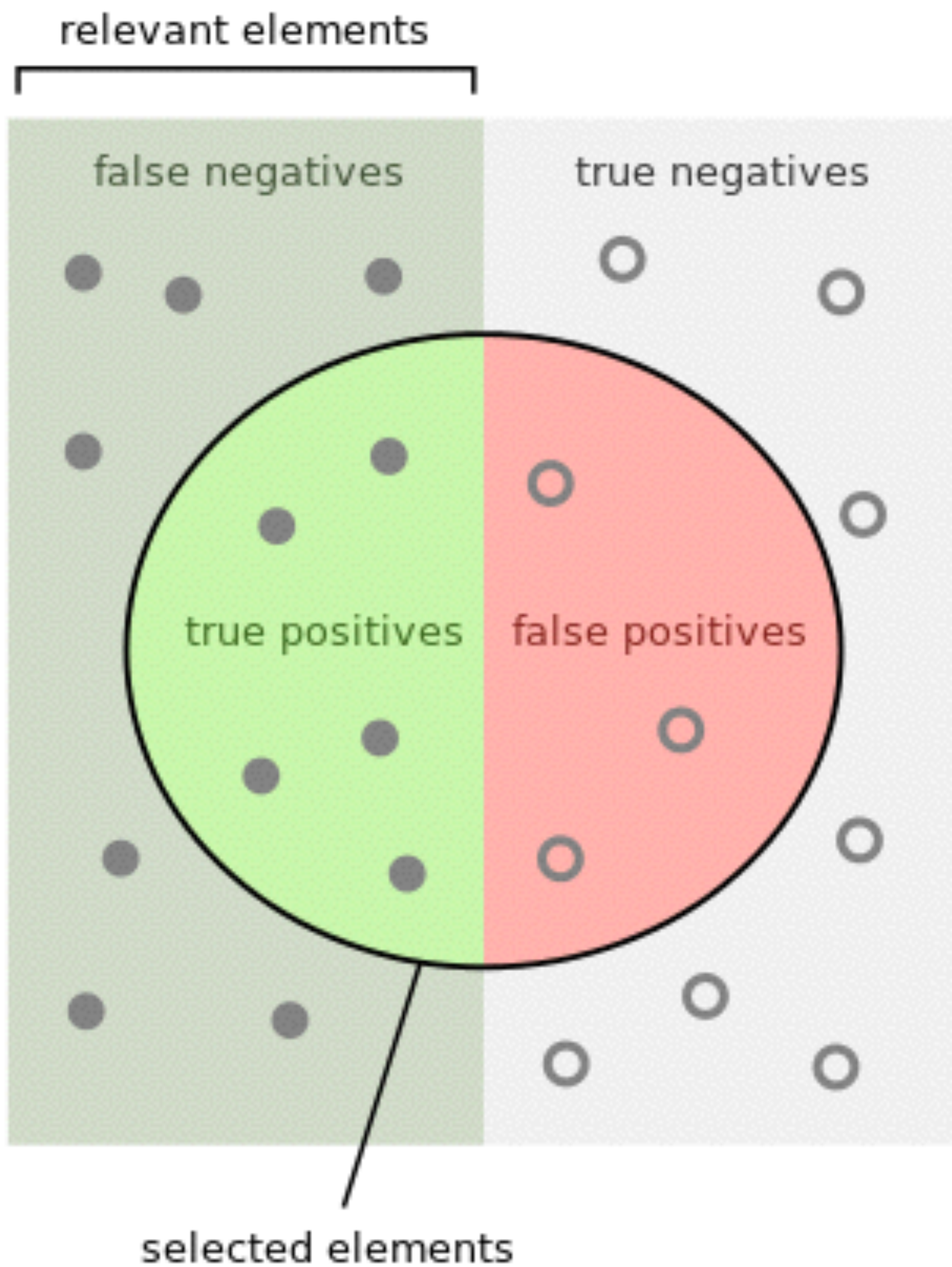
Precision = 0.847

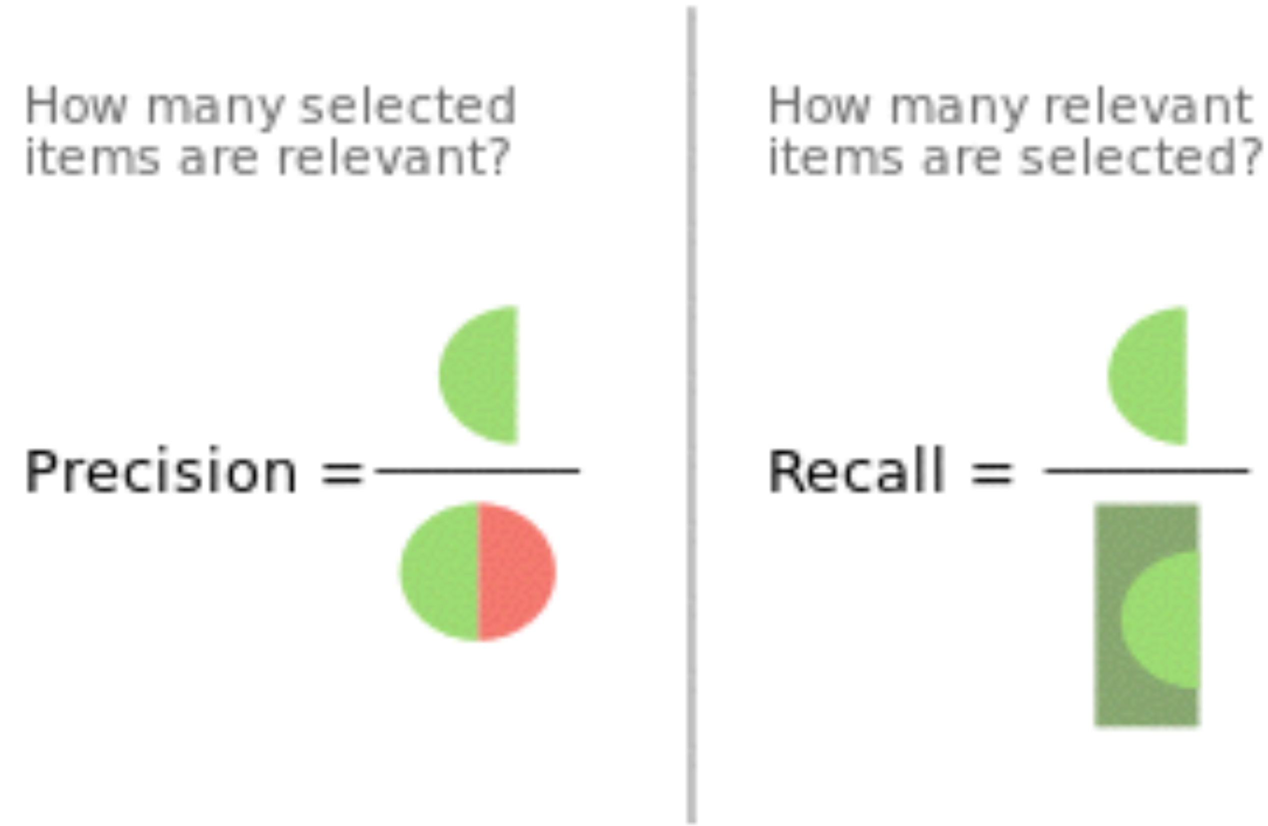
Recall = 0.630

F1-Score = 0.723

Source: Bo Han, Paul Cook and Timothy Baldwin
"Automatically Constructing a Normalisation Dictionary for Microblogs" EMNLP-CoNLL 2012

F-measure

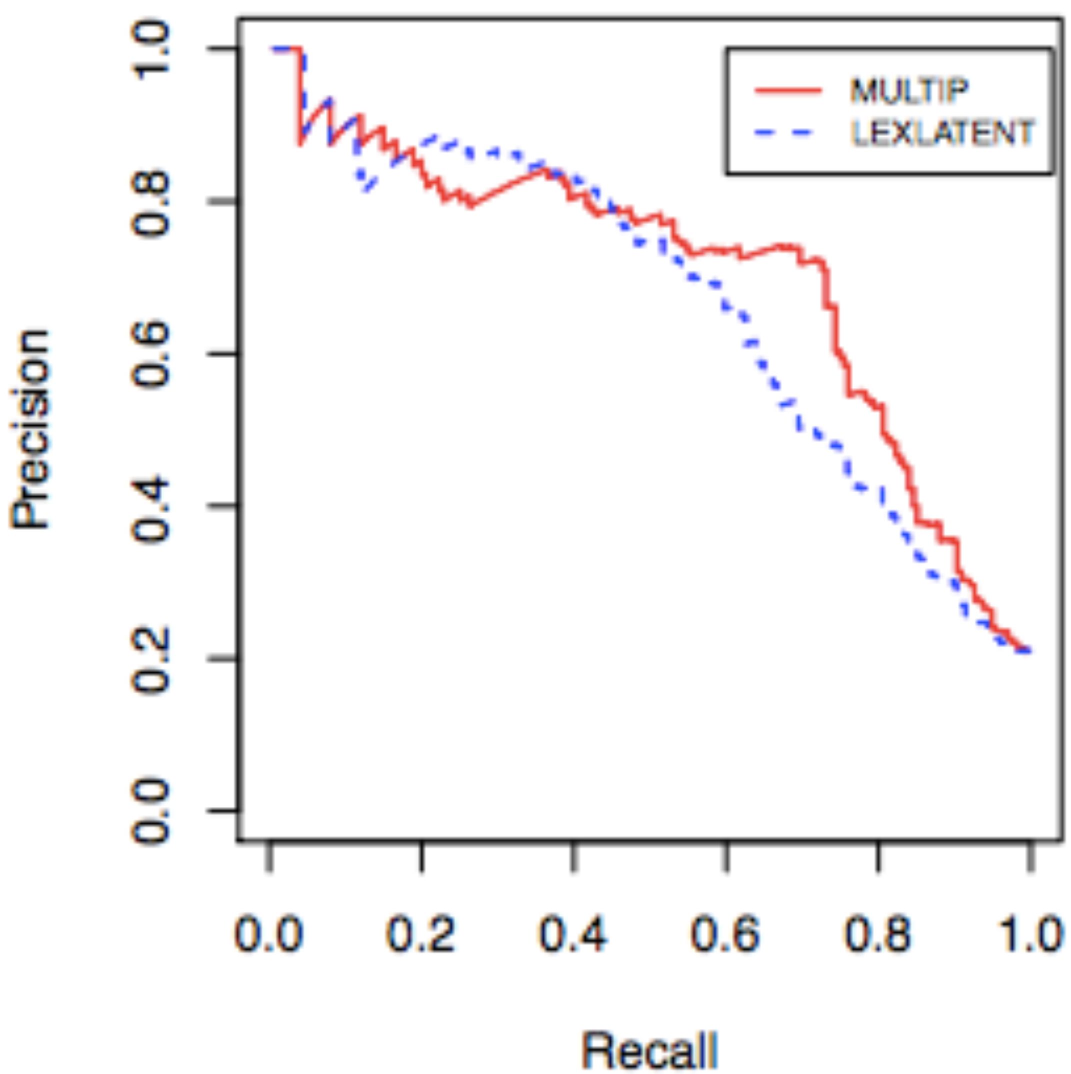


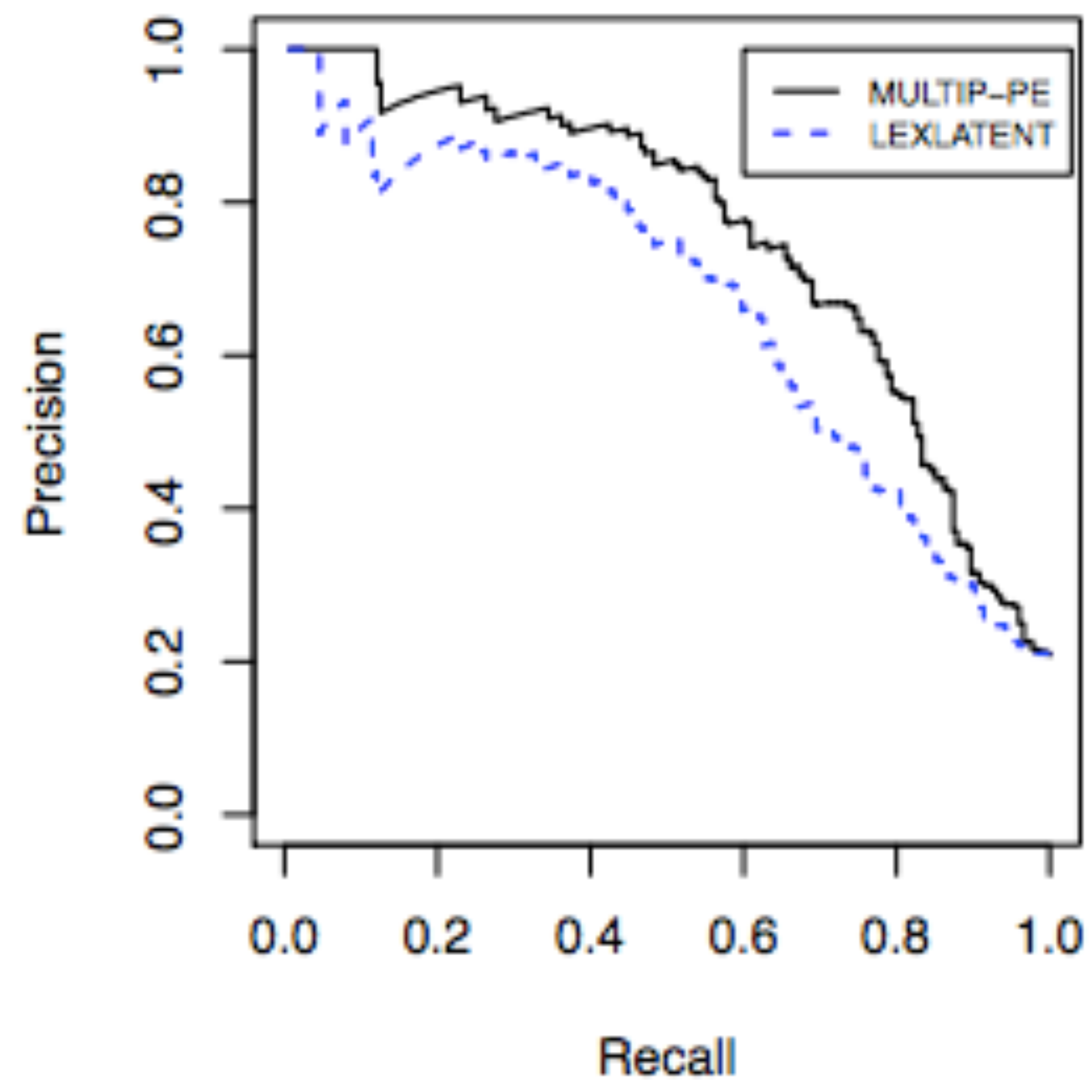


$$F_1 = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

Source: Wikipedia

Precision-Recall Curve





showing the trade-off, no threshold picking

Phrase-level Normalization

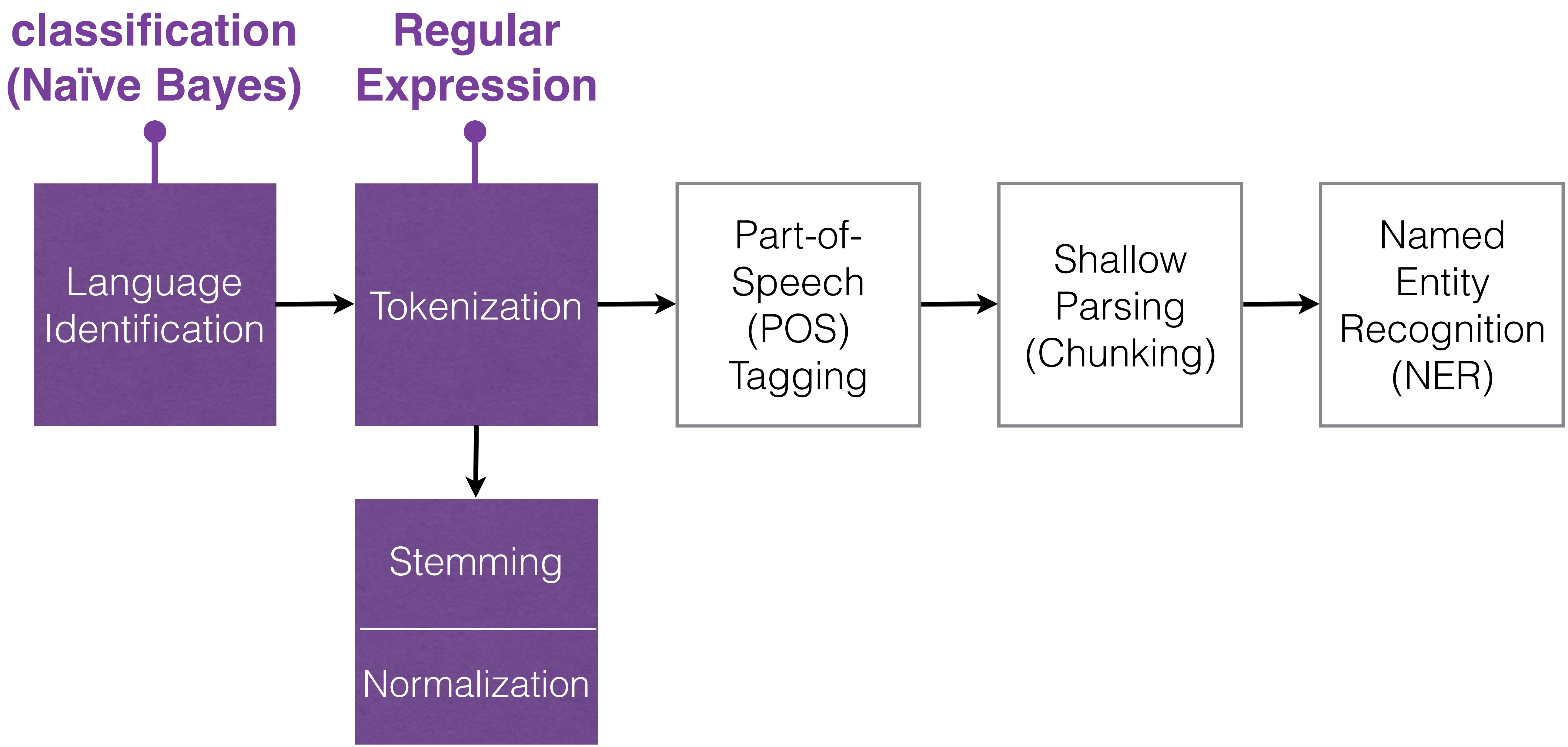
- word-level normalization is insufficient for many cases:

in-vocabulary words

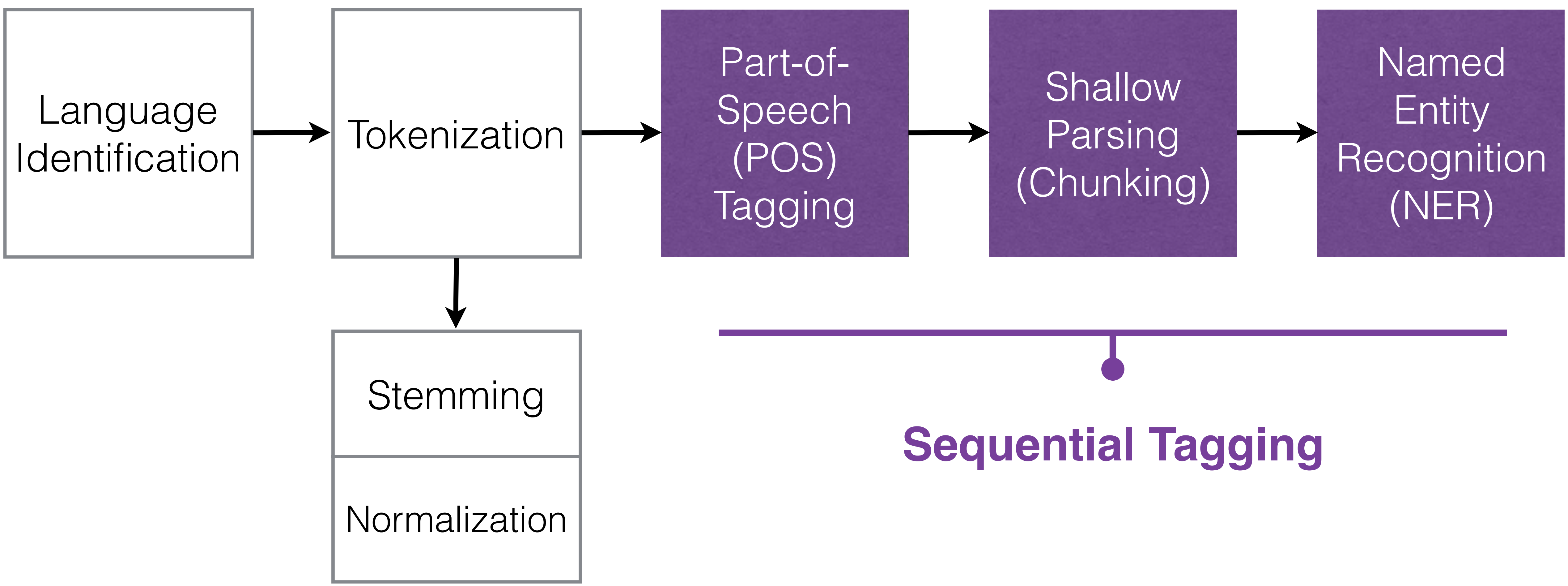
Category	Example
1-to-many	everytime → every time
incorrect IVs	can’t want for → can’t wait for
grammar	I’m going a movie → I’m going to a movie
ambiguities	4 → 4 / 4th / for / four

Source: Wei Xu, Alan Ritter, Ralph Grishman
“Gathering and Generating Paraphrases from Twitter with Application to Normalization” BUCC 2013

Summary


classification
(Naïve Bayes)
Regular
Expression
Language
Identification
Tokenization
Part-of-
Speech
(POS)
Tagging
Shallow
Parsing
(Chunking)
Named
Entity
Recognition
(NER)
Stemming
Normalization

Next Lecture



Thank You!

Follow @cocoweixu

Instructor: Wei Xu

www.cis.upenn.edu/~xwe/

Course Website: socialmedia-class.org