CS 4248
Natural Language Processing

Professor NG Hwee Tou
Department of Computer Science
School of Computing
National University of Singapore
nght@comp.nus.edu.sg

Basic Text Preprocessing

- Regular expressions, finite state automata
- Word tokenization and normalization

Regular Expressions

- Regular expression (RE): A formula (in a special language) for specifying a set of strings
- String: A sequence of alphanumeric characters (letters, digits, spaces, tabs, and punctuation symbols)

Regular Expression Patterns

- RE can be considered as a pattern to specify text search strings to search a corpus of texts
- Show the exact part of the string in a line that first matches a RE pattern

Regular Expression Patterns

RE	String matched
/woodchucks/	“interesting links to woodchucks and lemurs”
/a/	“Mary Ann stopped by Mona’s”
/Claire says,/	“My gift please,” Claire says,”
/song/	“all our pretty songs”
/!/	“Leave him behind!” said David

Regular Expression Patterns

RE	Match
`*`	zero or more occurrences of the previous char or expression
`+`	one or more occurrences of the previous char or expression
`?`	exactly zero or one occurrence of the previous char or expression
`{n}`	*n* occurrences of the previous char or expression
`{n,m}`	from *n* to *m* occurrences of the previous char or expression
`{n,}`	at least *n* occurrences of the previous char or expression
`.`	one occurrence of any character

Regular Expression Patterns

RE	Match	Example Patterns Matched
`woodchucks?`	woodchuck or woodchucks	"woodchuck"
`colou?r`	color or colour	"colour"

Regular Expression Patterns

- Python regular expression module (re)

```
import re
re.search("i.", "uninteresting")

return:

<re.Match object; span=(2, 4), match='in'>
```

Regular Expression Patterns

RE	Match	Example Patterns Matched
`\*`	an asterisk “*”	“K*A*P*L*A*N”
`\.`	a period “.”	“Dr. Livingston, I presume”
`\?`	a question mark	“Why don’t they come and lend a hand?”
`\n`	a newline	
`\t`	a tab	

Regular Expression Patterns

RE	Match	Example Patterns
`/[wW]oodchuck/`	Woodchuck or woodchuck	"Woodchuck"
`/[abc]/`	'a', 'b', *or* 'c'	"In uomini, in soldati"
`/[1234567890]/`	any digit	"plenty of 7 to 5"

Regular Expression Patterns

RE	Match	Example Patterns Matched
`/[A-Z]/`	an upper case letter	“we should call it ‘Drenched Blossoms’ ”
`/[a-z]/`	a lower case letter	“my beans were impatient to be hoed!”
`/[0-9]/`	a single digit	“Chapter 1: Down the Rabbit Hole”

RE	Match (single characters)	Example Patterns Matched
`[^A-Z]`	not an upper case letter	“Oyfn pripetchik”
`[^Ss]`	neither ‘S’ nor ‘s’	“I have no exquisite reason for’t”
`[^\.]`	not a period	“our resident Djinn”
`[e^]`	either ‘e’ or ‘^’	“look up ^ now”
`a^b`	the pattern ‘`a^b`’	“look up a^b now”

Regular Expression Patterns

Kleene *
/a*/ : zero or more a's

/aa*/ : one or more a's

/[ab]*/ : zero or more a's or b's (match strings like aaaa or abaab or bbbb)

Kleene +
/a+/ : one or more a's

Regular Expression Patterns

RE	Match	Example Patterns
`/beg.n/`	any character between *beg* and *n*	begin, beg'n, begun

/aardvark.*aardvark/
Match any string preceded by aardvark and ended by aardvark

Regular Expression Patterns

Anchors:

^ : start of a line
$: end of a line

/^The dog\.$/
Match a line that contains only “The dog.”

Regular Expression Patterns

Operator precedence hierarchy (highest to lowest)

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

Regular Expression Patterns

Disjunction
/cat|dog/ : matches “cat” or “dog”

Grouping
/gupp(y|ies)/ : matches “guppy” or “guppies”

Regular Expression Patterns

RE	Expansion	Match	Examples
\d	[0-9]	any digit	Party_of_5
\D	[^0-9]	any non-digit	Blue_moon
\s	[_\r\t\n\f]	white space (space, tab)	
\S	[^\s]	non-whitespace	in_Concord

Regular Expression Patterns

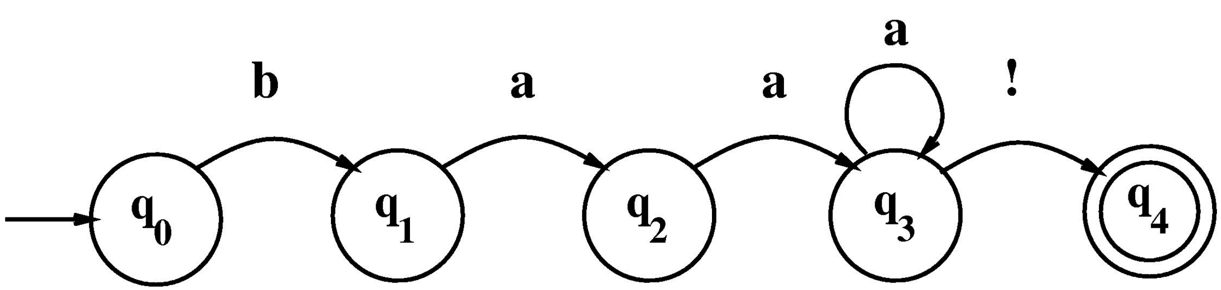
Find price or hardware or software:

/\$[0-9]+(\.[0-9][0-9])?/
/[0-9]+_*(GHz|[Gg]igahertz)/
/(Windows_*(11|10|8|7|Vista)?)/

Finite State Automata

- Regular expressions
- Finite state automata (FSA)

FSA



RE: /baa+!/

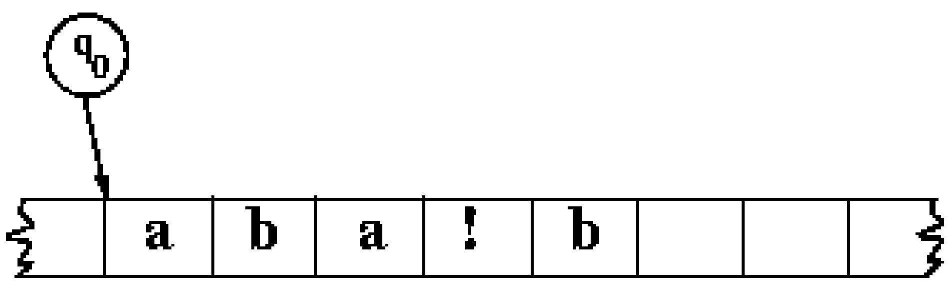
baa!
baaa!
baaaa!
baaaaa!
...

	Input		
State	b	a	!
0	1	∅	∅
1	∅	2	∅
2	∅	3	∅
3	∅	3	4
4:	∅	∅	∅

state-transition table

FSA

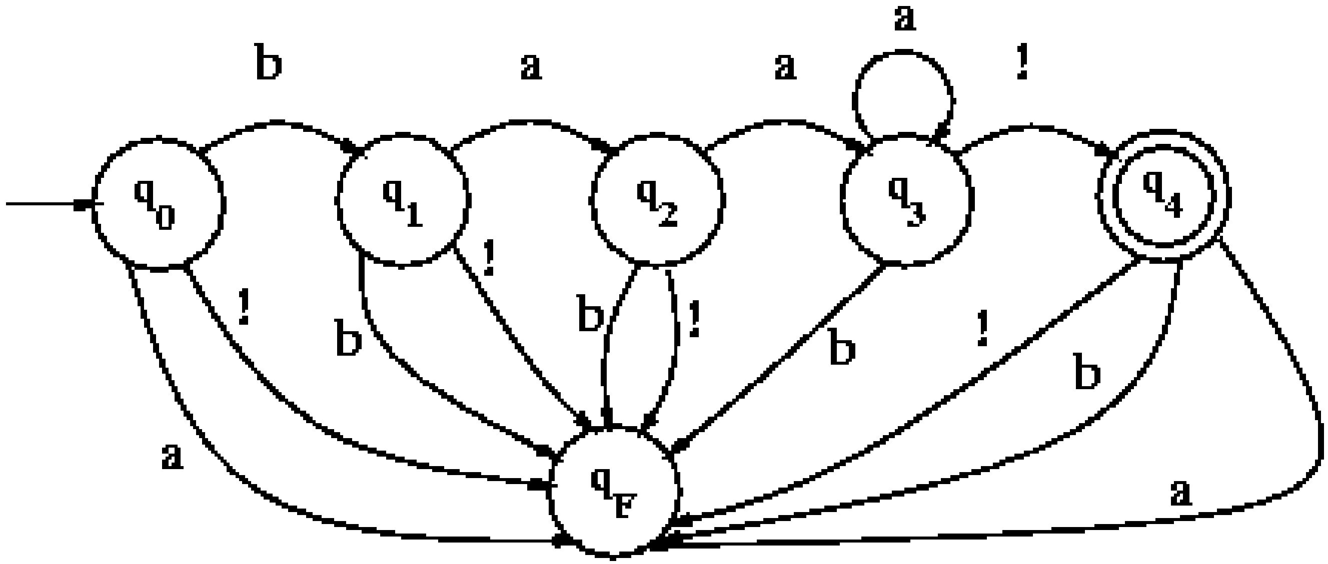
- Accepts an input string if we run out of input **and** the FSA is in an accepting state
- Rejects an input string otherwise



Formal Definition of FSA

- Q: a finite set of N states $q_0, q_1, \ldots, q_{N-1}$
- Σ: a finite input alphabet of symbols
- q_0: the start state
- F: the set of final states, $F \subseteq Q$
- $\delta(q,i)$: the transition function between states. Given a state $q \in Q$ and an input symbol $i \in \Sigma$, $\delta(q,i)$ returns a new state $q' \in Q$

Adding a Fail State



Formal Language

- Formal language
  - A formal language is a set of strings
  - A model which can both generate and recognize all and only the strings of a formal language acts as a *definition* of the formal language
  - Given a model m (e.g., a FSA), L(m) is the formal language characterized by m

Words and Corpora

- Corpora: computer-readable collections of text or speech
- Plural: corpora; Singular: corpus
- Which words to count

Types vs. Tokens

- Tokens (N): Total number of running words
- Types (B): Number of distinct words in a corpus (size of the vocabulary)
- Example: “They picnicked by the pool, then lay back on the grass and looked at the stars.”
  - 16 word tokens, 14 word types (not counting punctuation symbols)

Counting Words in Corpora

- Brown Corpus
  - 1 million words
  - 500 texts
  - Varied genres (newspaper, fiction, non-fiction, academic, etc.)
  - Assembled at Brown University in 1963-64
  - The first large on-line text collection used in corpus-based NLP research

How Many Words in English?

- Shakespeare's complete works
  - 884,647 wordform tokens
  - 29,066 wordform types
- Brown Corpus
  - 1 million wordform tokens
  - 61,805 wordform types
  - 37,851 lemma types

How Many Words in English?

- WordNet
  - Most widely used online English dictionary for NLP research (https://wordnet.princeton.edu/)
  - Store lemma of nouns, verbs, adjectives, adverbs
  - Include multi-word phrases (act_of_god, bamboo_shoot, iron_curtain, jumbo_jet, put_off, put_out)
  - 117,000 nouns
  - 11,000 verbs
  - 22,000 adjectives
  - 4,000 adverbs

Text Preprocessing

- Every NLP task requires text preprocessing
    1. Tokenizing (segmenting) words
    2. Normalizing words
    3. Segmenting sentences

Tokenization

- Segmenting a running text into words
- Breaking off punctuation symbols as separate tokens

Space-Based Tokenization

- A very simple way to tokenize
  - For languages that use space characters between words
    - Arabic, Greek, Latin, etc., based writing systems
  - Segment off a token between instances of spaces
- Unix tools for space-based tokenization
  - The "`tr`" command (stands for “translate”)
  - Inspired by Ken Church's UNIX for Poets
  - Given a text file, output the word tokens and their frequencies

Simple Tokenization in UNIX

- Given a text file, output the word tokens and their frequencies

```
tr 'A-Z' 'a-z' < shakes.txt | tr -sc 'A-Za-z' '\n' |
sort | uniq -c | sort -n -r
```

- To illustrate, suppose the file shakes.txt is:

```
low1 low Low low45 low lowest newEr newer newer
newer newer newer wider wider wIdeR new new
```

- Note that some characters are in upper case

Output of Each Step

```
tr 'A-Z' 'a-z' < shakes.txt
```

- Convert all upper case letters to lower case
- Output:

```
low1 low low low45 low lowest newer newer newer
newer newer newer wider wider wider new new
```

Output of Each Step

```
tr 'A-Z' 'a-z' < shakes.txt | tr -sc 'A-Za-z' '\n'
```

- Replace every non-letter (`-c` stands for complement) with a newline character (`\n`) and squeezes (`-s`) a sequence of newline characters into a single newline character
- Output on the next slide

Output of Each Step

```
low
low
low
low
low
lowest
newer
newer
newer
newer
newer
newer
wider
wider
wider
new
new
```

Output of Each Step

```
tr 'A-Z' 'a-z' < shakes.txt | tr -sc 'A-Za-z' '\n'
| sort
```

- Sort the list alphabetically
- Output shown on the right:

```
low
low
low
low
low
lowest
new
new
newer
newer
newer
newer
newer
newer
wider
wider
wider
```

Output of Each Step

```
tr 'A-Z' 'a-z' < shakes.txt | tr -sc 'A-Za-z' '\n'
| sort | uniq -c
```

- Keep only the unique tokens in the list and add their counts
- Output:

```
5 low
1 lowest
2 new
6 newer
3 wider
```

Output of Each Step

```
tr 'A-Z' 'a-z' < shakes.txt | tr -sc 'A-Za-z' '\n'
| sort | uniq -c | sort -n -r
```

- Sort the list in numerically (`-n`) decreasing (`-r`) order based on the counts
- Output

```
6 newer
5 low
3 wider
2 new
1 lowest
```

Issues in Tokenization

- Can't just blindly remove punctuation symbols
  - Ph.D., AT&T, m.p.h.
  - Prices ($45.55)
  - Dates (01/02/06)
  - URLs (https://www.comp.nus.edu.sg)
  - Hashtags (#nlproc)
  - Email addresses (nght@comp.nus.edu.sg)
- When should multiword expressions (MWE) be words?
  - New York, rock 'n' roll

Tokenization in Languages without Spaces

Many languages (like Chinese, Japanese, Thai) don't use spaces to separate words!

How do we decide where the token boundaries should be?

Word tokenization in Chinese

Chinese words are composed of characters called "**hanzi"** (or sometimes just "**zi**")

Each one represents a meaning unit called a morpheme.

Each word has on average 2.4 of them.

But deciding what counts as a word is complex and not agreed upon.

Word Tokenization in Chinese

- 姚明进入总决赛 “Yao Ming reaches the finals”

- 3 words?
- 姚明 进入 总决赛
- YaoMing reaches finals

- 5 words?
- 姚 明 进入 总 决赛
- Yao Ming reaches overall finals

Word Tokenization / Segmentation

So in Chinese, it's common to just treat each character (zi) as a token.

- So the **segmentation** step is very simple

In other languages (like Thai and Japanese), more complex word segmentation is required.

Another Option for Text Tokenization

Instead of

- white-space segmentation
- single-character segmentation

Use the data to tell us how to tokenize

Subword tokenization (because tokens can be parts of words as well as whole words)

Purpose: better handling of out-of-vocabulary words and rare words (e.g., Reaganomics, kaypohism, etc)

Subword Tokenization

- Byte-Pair Encoding (BPE)
- Two parts:
  – A token learner that takes a raw training corpus and induces a vocabulary (a set of tokens)
  – A token segmenter that takes a raw test sentence and tokenizes it according to that vocabulary

BPE Token Learner

Initialize vocabulary to the set of all individual characters: { A, B, C, D, …, a, b, c, d, … }

- Repeat:
  - Choose the two symbols that are most frequently adjacent in the training corpus (say ‘X', ‘Y')
  - Add a new merged symbol ‘XY' to the vocabulary
  - Replace every adjacent ‘X' ‘Y' in the corpus with ‘XY'
- Until *k* merges have been done or no more merging can be done

BPE Token Learner Algorithm

function BYTE-PAIR ENCODING(strings C, number of merges k) **returns** vocab V

$V \leftarrow$ all unique characters in C # initial set of tokens is characters
for $i = 1$ **to** k **do** # merge tokens til k times
$t_L, t_R \leftarrow$ Most frequent pair of adjacent tokens in C
$t_{NEW} \leftarrow t_L + t_R$ # make new token by concatenating
$V \leftarrow V + t_{NEW}$ # update the vocabulary
Replace each occurrence of t_L, t_R in C with t_{NEW} # and update the corpus
return V

Byte Pair Encoding (BPE)

Most subword algorithms are run inside space-separated tokens

So we commonly first add a special end-of-word symbol '__' before space in training corpus

Next, separate into letters.

BPE token learner

Original corpus:

low low low low low lowest lowest newer newer newer newer newer newer wider wider wider new new

Add end-of-word tokens, resulting in this vocabulary:

vocabulary

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

BPE token learner

corpus		**vocabulary**
5	l o w _	_, d, e, i, l, n, o, r, s, t, w
2	l o w e s t _	
6	n e w e r _	
3	w i d e r _	
2	n e w _	

Merge e r to er

corpus		**vocabulary**
5	l o w _	_, d, e, i, l, n, o, r, s, t, w, er
2	l o w e s t _	
6	n e w er _	
3	w i d er _	
2	n e w _	

BPE

corpus		**vocabulary**
5	l o w _	_, d, e, i, l, n, o, r, s, t, w, er
2	l o w e s t _	
6	n e w er _	
3	w i d er _	
2	n e w _	

Merge er _ to er_

corpus		**vocabulary**
5	l o w _	_, d, e, i, l, n, o, r, s, t, w, er, er_
2	l o w e s t _	
6	n e w er_	
3	w i d er_	
2	n e w _	

BPE

corpus		**vocabulary**
5	l o w _	_, d, e, i, l, n, o, r, s, t, w, er, er_
2	l o w e s t _	
6	n e w er_	
3	w i d er_	
2	n e w _	

Merge n e to ne

corpus		**vocabulary**
5	l o w _	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne
2	l o w e s t _	
6	ne w er_	
3	w i d er_	
2	ne w _	

BPE

The next merges are:

Merge	**Current Vocabulary**
`(ne, w)`	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new
`(l, o)`	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo
`(lo, w)`	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low
`(new, er_)`	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_
`(low, _)`	_, d, e, i, l, n, o, r, s, t, w, er, er_, ne, new, lo, low, newer_, low_
...	

BPE Token Segmenter Algorithm

On the test data, run each merge learned from the training data:

- Greedily
- In the order we learned them

So, merge every e r to er, then merge er _ to er_, etc.

- Result:
  - Test set "n e w e r _" would be tokenized as a full word
  - Test set "l o w e r _" would be two tokens: "low er_"

Properties of BPE tokens

Usually include frequent words

And frequent subwords, which are often morphemes like *-est* or *-er*

A morpheme is the smallest meaning-bearing unit of a language

- *unlikeliest* has 3 morphemes *un-*, *likely*, and *-est*

Text Normalization

- Converting text to a more convenient, standard form
- Expanding clitic contractions
  - Clitic: a part of a word that cannot stand on its own and can only occur when it is attached to another word
  - we're → we are; I've → I have

Word Normalization

- Putting words/tokens in a standard form
  - U.S. or US
  - uhhuh or uh-huh
  - Fed or fed
  - am, is, are, or be

Case Folding

- Applications like information retrieval (IR): reduce all letters to lower case
  - Since users tend to use lower case
  - Possible exception: upper case in mid-sentence?
    - Bush vs. bush
    - Fed vs. fed
    - He vs. he
- For sentiment analysis, machine translation, Information extraction
  - Case is helpful (US versus us is important)

Lemmatization

Represent all words as their lemma, their shared root (dictionary headword form):

- *am, is, are* $\rightarrow$ *be*
- *car, cars, car's*, *cars'* $\rightarrow$ *car*
- *He is reading detective stories* $\rightarrow$ *He be read detective story*

Morphology

- Morpheme: the minimal meaning-bearing unit in a language
- 2 classes of morphemes: stems & affixes
- Stems: The core meaning-bearing units
- Affixes: Parts that adhere to stems, often with grammatical functions
- Morphology: The study of the way words are built up from morphemes
- Word formation and structure of words

Morphology

- Example: The word “cats” is made up of 2 morphemes: “cat” (stem) and “-s” (affix)
- Other examples:
  – fox, foxes
  – eat, ate, eats, eating, eaten
  – kill, killer
  – clue, clueless

Morphology

- Types of affixes
  - prefix: unbuckle (stem: buckle, prefix: un-)
  - suffix: eats (stem: eat, suffix: -s)
- A word can have more than one affix
  - rewrites: write, re-, -s
  - unbelievably: believe, un-, -able, -ly

Why Morphology

- Listing all the different morphological variants of a word in a dictionary is inefficient
- Affixes are *productive*; they apply to new words (e.g., fax and faxing)
- For morphologically complex languages like Turkish, it is impossible to list all morphological variants of every word

The Porter Stemming Algorithm

- Stemming: simple version of morphological analysis by stripping off affixes
- Porter stemmer: A simple and efficient stemming algorithm used in information retrieval
- A series of rewrite rules run in a cascade; output of each pass is fed as input to the next pass
- Does not require a lexicon
- https://tartarus.org/martin/PorterStemmer/

The Porter Stemming Algorithm

- A series of rewrite rules:

  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)

- Reduce terms to stems, chopping off affixes crudely

The Porter Stemming Algorithm

Input text:

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.

Stemmed output text:

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

Penn Treebank Tokenization Standard

- Separate out clitics
  - doesn't → does n't
  - John's → John 's
- Keep hyphenated words together
- Separate out all punctuation symbols
- 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 ” .`

Sentence Segmentation

!, ? mostly unambiguous, but period "." is very ambiguous

- Sentence boundary
- Abbreviations like Inc. or Dr.
- Numbers like .02% or 4.3

Common algorithm: Use rules or machine learning to classify a period as either (a) part of the word or (b) a sentence-boundary.

- An abbreviation dictionary can help

Sentence segmentation can then often be done by rules