Lexical Analysis(NLP)

Text Normalization

We'll then turn to a set of tasks collectively called text normalization. Normalizing text means converting it to a more convenient, standard form.

- **Tokenization**: separating out or tokenizing words from running text
- **Lemmatization**: the task of determining that two words have the same root, despite their surface differences
- **Stemming** refers to a simpler version of lemmatization in which we mainly just strip suffixes from the end of the word.
- **Sentence segmentation**: breaking up a text into individual sentences, using cues like periods or exclamation points.
- We'll introduce a metric called **Edit Distance** that measures how similar two strings are based on the number of edits (insertions, deletions, substitutions) it takes to change one string into the other.

Regular Expressions

- One of the most useful tools for text processing in computer science has been the **regular expression** (often shortened to **regex**).
- It is a language for specifying text search strings.
- Formally, a regular expression is an algebraic notation for characterizing a set of strings.
- A regular expression search function will search through the corpus (a single document or a collection) , returning all texts that match the pattern.
- Regular expressions come in many variants. We'll be describing extended regular expressions

Regular expressions are used everywhere

- Part of every text processing task
  - Not a general NLP solution
  - But very useful as part of those systems (e.g., for pre-processing or text formatting)
- Necessary for data analysis of text data
- A widely used tool in industry and academics

Basic Regular Expression Patterns

If you like to search a string, let's say "Bunny", you can do that by /Bunny/.

Regular expressions are case sensitive; lower case /s/ is distinct from upper case /S/ (/s/ matches a lower case s but not an upper case S).

This means that the pattern /woodchucks/will not match the string Woodchucks.

We can solve this problem with the use of the square braces [and].

The string of characters inside the braces specifies a **disjunction** of characters to match.

Regex	Example Patterns Matched
`/woodchucks/`	"interesting links to woodchucks and lemurs"
`/a/`	"Mary Ann stopped by Mona's"
`/!/`	"You've left the burglar behind again!" said Nori

Regex	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 Expressions: Ranges

- The regular expression /[1234567890]/specifies any single digit.
- The regular expression /[ABCDEFGHIJKLMNOPQRSTUVWXYZ]/ matches with any capital letter.
- In cases where there is a well-defined sequence associated with a set of characters, the brackets can be used with the dash (-) to specify any one character in a range.

Ranges using the dash:

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 the Rabbit Hole`

Regular Expressions: Negation in Disjunction

Carat as first character in [] negates the list

- Note: Carat means negation only when it's first in []
- Special characters (., *, +, ?) lose their special meaning inside []

Pattern	Matches	Examples
[^A-Z]	Not an upper case letter	Oyfn pripetchik
[^Ss]	Neither ‘S’ nor ‘s’	I have no exquisite reason”
[^.]	Not a period	Our resident Djinn
[e^]	Either e or ^	Look up ^ now

Regular Expressions: Convenient aliases

Pattern	Expansion	Matches	Examples
`\d`	`[0-9]`	Any digit	`Fahreneit 451`
`\D`	`[^0-9]`	Any non-digit	`Blue Moon`
`\w`	`[a-ZA-Z0-9_]`	Any alphanumeric or _	`Daiyu`
`\W`	`[^\w]`	Not alphanumeric or _	`Look!`
`\s`	`[ \r\t\n\f]`	Whitespace (space, tab)	`Look␣up`
`\S`	`[^\s]`	Not whitespace	`Look up`

Regular Expressions: More Disjunction

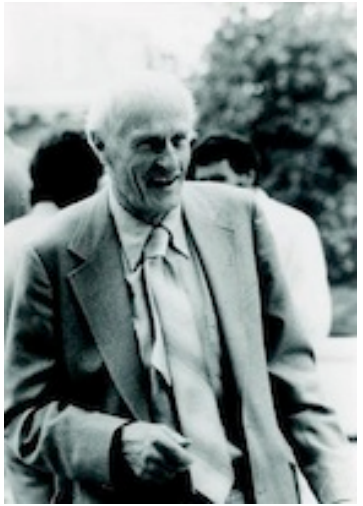
Groundhog is another name for woodchuck!

The pipe symbol | for disjunction

Pattern	Matches
`groundhog|woodchuck`	woodchuck
`yours|mine`	yours
`a|b|c`	= [abc]
`[gG]roundhog|[Ww]oodchuck`	Woodchuck

Wildcards, optionality, repetition: . ? * +

Pattern	Matches	Examples
`beg.n`	Any char	begin begun beg3n beg n
`woodchucks?`	Optional s	woodchuck woodchucks
`to*`	0 or more of previous char	t to too tooo
`to+`	1 or more of previous char	to too tooo toooo



Stephen C Kleene

Kleene *, Kleene +

Regular Expressions: Anchors ^ $

Pattern	Matches
/^The/	Matches a line starts with “The”
^[A-Z]	Palo Alto
^[^A-Za-z]	1 “Hello”
\.$	The end.
.$	The end? The end!
/^The dog\.$/	Matches a line that contains only the phrase “The dog.”

Regex	Match
^	start of line
$	end of line
\b	word boundary
\B	non-word boundary

\b matches a word boundary, and \B matches a non word-boundary. Thus, /\bthe\b/ matches the word “the” but not the word “other”.

precedence

- How can we specify both guppy and guppies? We cannot simply say /guppy|ies/, because that would match only the strings guppy and ies.
- This is because sequences like guppy take precedence over the disjunction operator |.
- To make the disjunction operator apply only to a specific pattern, we need to use the parenthesis operators (and).
- So the pattern /gupp(y|ies)/ would specify that we meant the disjunction only to apply to the suffixes y and ies.

- Kleene* operator applies by default only to a single character, not to a whole sequence.
- Suppose we want to match repeated instances of a string.
- Perhaps we have a line that has column labels of the form Column 1 Column 2 Column 3.
- The expression /Column␣[0-9]+␣*/ will not match any number of columns; instead, it will match a single column followed by any number of spaces!
- With the parentheses, we could write the expression /(Column␣[0-9]+␣*)*/ to match the word Column, followed by a number and optional spaces, the whole pattern repeated zero or more times.

This idea that one operator may take precedence over another, requiring us to sometimes use parentheses to specify what we mean, is formalized by the operator precedence hierarchy for regular expressions.

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

The iterative process of writing regex's

Find me all instances of the word “the” in a text.

`the`

Misses capitalized examples

`[tT]he`

Incorrectly returns `other` or `Theology`

`\W[tT]he\W`

False positives and false negatives

The process we just went through was based on fixing two kinds of errors:

1. Not matching things that we should have matched (The)

   False negatives

2. Matching strings that we should not have matched (there, then, other)

   False positives

Characterizing work on NLP

In NLP we are always dealing with these kinds of errors.

Reducing the error rate for an application often involves two antagonistic efforts:

- Increasing coverage (or *recall*) (minimizing false negatives).
- Increasing accuracy (or *precision*) (minimizing false positives)

Regular expressions play a surprisingly large role

Widely used in both academics and industry

1. Part of most text processing tasks, even for big neural language model pipelines
   - including text formatting and pre-processing
2. Very useful for data analysis of any text data

Basic Text Processing

More Regular Expressions: Substitutions and ELIZA

Substitutions

Substitution in Python and UNIX commands:

```
s/regexp1/pattern/
```

e.g.:

```
s/colour/color/
```

Capture Groups

- Say we want to put angles around all numbers:

  the 35 boxes → *the <35> boxes*

- Use parethesis () to "capture" a pattern into a numbered register (1, 2, 3…)
- Use \1 to refer to the contents of the register

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

Capture groups: multiple registers

```
/the (.*)er they (.*), the \1er we \2/
```

Matches

the faster they ran, the faster we ran

But not

the faster they ran, the faster we ate

But suppose we don't want to capture?

Parentheses have a double function: grouping terms, and capturing

Non-capturing groups: add a ?: after paren:

```
/(?:some|a few) (people|cats) like some \1/
```

matches

- `some cats like some cats`

but not

- `some cats like some some`

Lookahead assertions

`(?= pattern)` is true if pattern matches, but is **zero-width; doesn't advance character pointer**

`(?! pattern)` true if a pattern does not match

How to match, at the beginning of a line, any single word that doesn’t start with “Volcano”:

```
/^(?!Volcano)[A-Za-z]+/
```

Simple Application: ELIZA

Early NLP system that imitated a Rogerian psychotherapist

- Joseph Weizenbaum, 1966.

Uses pattern matching to match, e.g.,:

- "I need X"

and translates them into, e.g.

- "What would it mean to you if you got X?

Simple Application: ELIZA

Men are all alike.
IN WHAT WAY

They're always bugging us about something or other.
CAN YOU THINK OF A SPECIFIC EXAMPLE

Well, my boyfriend made me come here.
YOUR BOYFRIEND MADE YOU COME HERE

He says I'm depressed much of the time.
I AM SORRY TO HEAR YOU ARE DEPRESSED

How ELIZA works

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?/

How many words in a sentence?

"I do uh main- mainly business data processing"

- Fragments, filled pauses

"Seuss's cat in the hat is different from other cats!"

- **Lemma**: same stem, part of speech, rough word sense
  - cat and cats = same lemma
- **Wordform**: the full inflected surface form
  - cat and cats = different wordforms

How many words in a sentence?

they lay back on the San Francisco grass and looked at the stars and their

Type: an element of the vocabulary.

Token: an instance of that type in running text.

How many?

- 15 tokens (or 14)
- 13 types (or 12) (or 11?)

How many words in a corpus?

N = number of tokens

V = vocabulary = set of types, ***|V|*** is size of vocabulary

Heaps Law = Herdan's Law = $|V| = kN^{\beta}$ where often $0.67 < \beta < 0.75$

i.e., vocabulary size grows with > square root of the number of word tokens

	Tokens = N	Types = |V|
Switchboard phone conversations	2.4 million	20 thousand
Shakespeare	884,000	31 thousand
COCA	440 million	2 million
Google N-grams	1 trillion	13+ million

Corpora

Words don't appear out of nowhere!

A text is produced by

- a specific writer(s),
- at a specific time,
- in a specific variety,
- of a specific language,
- for a specific function.

Corpora vary along dimension like

- **Language**: 7097 languages in the world
- **Variety**, like African American English varieties.
  - AAE Twitter posts might include forms like "*iont" (I don't)*
- **Code switching**, e.g., Spanish/English, Hindi/English:

  S/E: Por primera vez veo a @username actually being hateful! It was beautiful:)

  [For the first time I get to see @username actually being hateful! it was beautiful:)]

  H/E: dost tha or ra- hega ... dont wory ... but dherya rakhe

  [“he was and will remain a friend ... don’t worry ... but have faith”]
- **Genre:** newswire, fiction, scientific articles, Wikipedia
- **Author Demographics**: writer's age, gender, ethnicity, SES

Corpus **datasheets**

Gebru et al (2020), Bender and Friedman (2018)

Motivation:

- Why was the corpus collected?
- By whom?
- Who funded it?

Situation: In what situation was the text written?

Collection process: If it is a subsample how was it sampled? Was there consent? Pre-processing?

+**Annotation process, language variety, demographics, etc.**

Basic Text Processing

Word tokenization

Text Normalization

Every NLP task requires text normalization:

1. Tokenizing (segmenting) words
2. Normalizing word formats
3. Segmenting sentences

Space-based tokenization

A very simple way to tokenize

- For languages that use space characters between words
  - Arabic, Cyrillic, Greek, Latin, etc., based writing systems
- Segment off a token between instances of spaces

Unix tools for space-based tokenization

- The "tr" command
- Inspired by Ken Church's UNIX for Poets
- Given a text file, output the word tokens and their frequencies

Simple Tokenization in UNIX

(Inspired by Ken Church's UNIX for Poets.)

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

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

Change all non-alpha to newlines

Sort in alphabetical order

Merge and count each type

```
1945 A
  72 AARON
  19 ABBESS
   5 ABBOT
 ... ...
```

```
25 Aaron
 6 Abate
 1 Abates
 5 Abbess
 6 Abbey
 3 Abbot
... ...
```

The first step: tokenizing

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

```
THE
SONNETS
by
William
Shakespeare
From
fairest
creatures
We
...
```

The second step: sorting

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

```
A
A
A
A
A
A
A
A
A
...
```

More counting

Merging upper and lower case

```
tr ‘A-Z’ ‘a-z’ < shakes.txt | tr –sc ‘a-z’ ‘\n’ | sort | uniq –c
```

Sorting the counts

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

```
23243 the
22225 i
18618 and
16339 to
15687 of
12780 a
12163 you
10839 my
10005 in
```

Issues in Tokenization

Can't just blindly remove punctuation:

- m.p.h., Ph.D., AT&T, cap’n
- prices ($45.55)
- dates (01/02/06)
- URLs (http://www.stanford.edu)
- hashtags (#nlproc)
- email addresses (someone@cs.colorado.edu)

Clitic: a word that doesn't stand on its own

- "are" in we're, French "je" in j'ai, "le" in l'honneur

When should multiword expressions (MWE) be words?

- New York, rock ’n’ roll

Tokenization in NLTK

Bird, Loper and Klein (2009), *Natural Language Processing with Python*. O'Reilly

```
>>> text = 'That U.S.A. poster-print costs $12.40...'
>>> pattern = r'''(?x)    # set flag to allow verbose regexps
...     ([A-Z]\.)+        # abbreviations, e.g. U.S.A.
...   | \w+(-\w+)*        # words with optional internal hyphens
...   | \$?\d+(\.\d+)?%?  # currency and percentages, e.g. $12.40, 82%
...   | \.\.\.            # ellipsis
...   | [][.,;"'?():-_`]  # these are separate tokens; includes ], [
... '''
>>> nltk.regexp_tokenize(text, pattern)
['That', 'U.S.A.', 'poster-print', 'costs', '$12.40', '...']
```

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.

How to do word tokenization in Chinese?

姚明进入总决赛“Yao Ming reaches the finals”

How to do 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

How to do 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

How to do 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

7 characters? (don't use words at all):
姚 明 进 入 总 决 赛
Yao Ming enter enter overall decision game

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.

- The standard algorithms are neural sequence models trained by supervised machine learning.

Basic Text Processing

Byte Pair Encoding

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)

Subword tokenization

Three common algorithms:

- **Byte-Pair Encoding (BPE)** (Sennrich et al., 2016)
- **Unigram language modeling tokenization** (Kudo, 2018)
- **WordPiece** (Schuster and Nakajima, 2012)

All have 2 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

Byte Pair Encoding (BPE) token learner

Let vocabulary be 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 'A', 'B')
- Add a new merged symbol 'AB' to the vocabulary
- Replace every adjacent 'A' 'B' in the corpus with 'AB'.

Until *k* merges have been 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) Addendum

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 (very fascinating🙄) 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

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

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
- (test frequencies don't play a role)

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*

WordPiece

- **WordPiece** is a **subword-based tokenization algorithm** used in models like **BERT**, originally developed by Google for machine translation.

- It builds a vocabulary of **subword units** and tokenizes text into these units for **efficient representation** of known and unknown words.

WordPiece – Training Phase

1. **Initialize** vocabulary with all individual characters (plus special tokens).
2. Treat each word in the training corpus as a **sequence of characters** plus an end-of-word marker.
   E.g., "playing" → ["p", "l", "a", "y", "i", "n", "g", "</w>"]
3. Count frequencies of all adjacent symbol pairs.
4. Merge the **most frequent pair** into a new symbol and update all word representations.
5. Repeat steps 3–4 until the vocabulary reaches the desired size (e.g., 30,000 tokens).

This is similar to **Byte-Pair Encoding (BPE)**, but WordPiece selects merges based on **likelihood maximization**, not raw frequency.

Example

Given a toy corpus:

unrelated

unwanted

unhappy

undo

Start with character-level tokens (all words split into characters + </w> marker):

[u n r e l a t e d </w>]

[u n w a n t e d </w>]

[u n h a p p y </w>]

[u n d o </w>]

Count adjacent pairs:

u n: 4 times

n w: 1

n h: 1

n d: 1

...

Merge most frequent pair: u n → un

Update corpus:

[un r e l a t e d </w>]

[un w a n t e d </w>]

[un h a p p y </w>]

[un d o </w>]

Repeat: Merge un h → unh, then h a → ha, etc.

Eventually you get vocabulary entries like:

un, unh, happy, ed, want, do, ##ed, ##ness, ##ing

What Is Likelihood Maximization in WordPiece?

Let’s assume we have a corpus of words and an initial vocabulary of individual characters.

The goal is to **merge subword pairs** in a way that **maximizes the probability of the training corpus** using a unigram language model, where:

Each word is represented as a sequence of subwords from the current vocabulary.

The probability of a word is the product of the probabilities of its subwords.

At each step:

Compute the likelihood of the corpus under the current vocabulary.

For every possible pair of symbols, simulate merging it.

Choose the merge that results in the **largest increase in likelihood**.

Mathematical Intuition

Suppose we tokenize word $w = t_1, t_2, ..., t_k$ using subwords.

The **likelihood** of the corpus is:

$L=\prod_{w\in Corpus}P(w)=\prod_{w\in Corpus}P(w)\prod_{t_i\in w}P(t_i)$

The goal is to choose a merge (say, merge A + B → AB) that **maximizes** this likelihood.

We do:

For each candidate merge, recompute tokenization of all words.

Compute the updated probabilities of subwords.

Pick the merge that yields the highest increase in total log-likelihood:

$\Delta \log L = \log L_{after} - \log L_{before}$

Illustration with an example

Corpus = ["unaffable", "unavoidable"]

Initial vocabulary: characters + special token </w>

Tokenized as:

u n a f f a b l e </w>

u n a v o i d a b l e </w>

Step 1: Count token sequences

All characters are tokens initially.

We want to merge pairs like:

a f, f f, u n, a b, a v, etc.

Suppose:

aff occurs more often than avo, but merging aff results in a smaller likelihood improvement than avo, because avo forms a frequently used morpheme.

Step 2: Simulate a merge: a v → av

Before merge:

unavoidable → u n a v o i d a b l e

After merge:

unavoidable → u n av o i d a b l e

If av is common in other words too, it will increase frequency → boosting its probability → increasing overall corpus likelihood.

Step 3: Compute probabilities

Let's suppose:

Before merge: P(a) = 0.1, P(v) = 0.05

After merge: P(av) = 0.12

If the number of tokens used to represent the corpus decreases and high-probability tokens increase, the overall log-likelihood goes **up**.

Step 4: Choose the best merge

Repeat this for all possible merges, and choose the one that **maximizes log-likelihood**.

Basic Text Processing

Word Normalization and other issues

Word Normalization

Putting words/tokens in a standard format

- U.S.A. or USA
- uhhuh or uh-huh
- Fed or fed

Case folding

Applications like IR: reduce all letters to lower case

- Since users tend to use lower case
- Possible exception: upper case in mid-sentence?
  - e.g., ***General Motors***
  - ***Fed*** vs. ***fed***
  - ***SAIL*** vs. ***sail***

For sentiment analysis, MT, Information extraction

- Case is helpful (***US*** versus ***us*** is important)

Lemmatization

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

- *am, are, is* → *be*
- *car, cars, car's, cars'* → *car*
- Spanish quiero (‘I want’), quieres (‘you want’)
  → querer ‘want'
- *He is reading detective stories*
  → *He be read detective story*

Porter Stemmer

Based on a series of rewrite rules run in series

- A cascade, in which output of each pass fed to next pass

Some sample rules:

$$\begin{aligned}
\text{ATIONAL} &\rightarrow \text{ATE} \quad \text{(e.g., relational} \rightarrow \text{relate)} \\
\text{ING} &\rightarrow \epsilon \quad \text{if stem contains vowel (e.g., motoring} \rightarrow \text{motor)} \\
\text{SSES} &\rightarrow \text{SS} \quad \text{(e.g., grasses} \rightarrow \text{grass)}
\end{aligned}$$

Dealing with complex morphology is necessary for many languages

- e.g., the Turkish word:
- Uygarlastiramadiklarimizdanmissinizcasina
- `(behaving) as if you are among those whom we could not civilize'
- Uygar `civilized' + las `become'

  + tir `cause' + ama `not able'

  + dik `past' + lar 'plural'

  + imiz 'p1pl' + dan 'abl'

  + mis 'past' + siniz '2pl' + casina 'as if'

Sentence Segmentation

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

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

Common algorithm: Tokenize first: use rules or ML 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 based on this tokenization.

Morphological Analysis

- Human language is a complicated thing.
- We use it to express our thoughts and we receive information and infer its meaning.
- Trying to understand language all together is not a viable approach.
- Linguists have developed whole disciplines that look at language from different perspectives and at different levels of detail.
- The point of morphology, for instance, is to study the variable forms and functions of words
- Words are defined in most languages as the smallest linguistic units that can form a complete utterance by themselves.
- The minimal parts of words that deliver aspects of meaning to them are called morphemes.

Lexemes

- By the term word, we often denote not just the one linguistic form in the given context but also the concept behind the form and the set of alternative forms that can express it. Such sets are called lexemes or lexical items, and they constitute the lexicon of a language.
- Lexemes can be divided by their behaviour into the lexical categories of verbs, nouns, adjectives, conjunctions, particles, or other parts of speech.
- The citation form of a lexeme, by which it is commonly identified, is also called its **lemma**.
- When we convert a word into its other forms, such as turning the singular mouse into the plural mice, we say we **inflect the lexeme**.
- When we transform a lexeme into another one that is morphologically related, regardless of its lexical category, we say we **derive** the lexeme: for instance, the nouns receiver and reception are derived from the verb receive.

Morphemes

- Morphological theories differ on whether and how to associate the properties of word forms with their structural components.

- These components are usually called segments or morphs.

- The morphs that by themselves represent some aspect of the meaning of a word are called morphemes of some function.

- Human languages employ a variety of devices by which morphs and morphemes are combined into word forms.

Morphology

- Morphology is the domain of linguistics that analyses the internal structure of words.
- Morphological analysis – exploring the structure of words
- Words are built up of minimal meaningful elements called morphemes: played = play-ed cats = cat-s unfriendly = un-friend-ly
- Two types of morphemes:

I. Stems: play, cat, friend

II. Affixes: -ed, -s, un-, -ly

- Two main types of affixes:

I. Prefixes precede the stem: un

II. Suffixes follow the stem: -ed, -s, un-, -ly

Stemming = find the stem by stripping off affixes

play = play replayed = re-play-ed computerized = comput-er-ize-d

Morphological Analysis in NLP

Purpose: To understand the structure of words and their components.

Key Tasks:

- **Stemming:** Reducing words to their root form (e.g., "running" to "run").
- **Lemmatization:** Converting words to their dictionary form (lemma), considering the context (e.g., "better" to "good").
- **Morpheme Boundary Detection:** Identifying where morphemes begin and end within a word.
- **Morphological Feature Tagging:** Assigning grammatical features (e.g., tense, number, gender) to morphemes.
- **Morphological Reinflection:** Generating different forms of a word based on inflectional rules.

Morphological Typology

- Morphological typology is a way of classifying the languages of the world that groups languages according to their common morphological structures.
- The field organizes languages on the basis of how those languages form words by combining morphemes.
- The morphological typology classifies languages into two broad classes of synthetic languages and analytical languages.
- Analytic languages contain very little inflection, instead relying on features like word order and auxiliary words to convey meaning.
- Synthetic languages, ones that are not analytic, are divided into two categories: agglutinative and fusional languages.
- Agglutinative languages rely primarily on discrete particles(prefixes, suffixes, and infixes) for inflection, ex: inter+national = international, international+ize = internationalize.
- While fusional languages "fuse" inflectional categories together, often allowing one word ending to contain several categories, such that the original root can be difficult to extract (anybody, newspaper).

Morpheme:

Definition: The smallest unit of language that carries meaning.

Types:

- Free morphemes: Can stand alone as a word (e.g., "cat", "run").
- Bound morphemes: Must be attached to other morphemes (e.g., prefixes like "un-" or suffixes like "-ed").

Examples: "unhappy" has two morphemes: "un-" (bound) and "happy" (free).

Lexeme:

Definition: A word and all its inflected forms.

Examples: The lexeme "RUN" includes the forms "run", "runs", "ran", and "running". The lexeme "HOPE" includes "hope", "hopes", "hoped", and "hoping".

Issues and Challenges

Morphology, the study of word structure, faces challenges in handling irregularities, ambiguities, and productivity in language. Here's a more detailed breakdown:

1. Irregularity:

- Many languages have irregular verb forms (e.g., "went," "ran") and noun plurals (e.g., "children") that don't follow standard morphological rules.

- This makes it difficult for computational systems to accurately process and generate these forms.

- Irregularities also exist in other word classes and can be lexically specific.

2. Ambiguity:

- Ambiguity can occur at different levels of morphology. For example, a single morpheme (the smallest meaningful unit of language) can have multiple meanings (e.g., the English plural marker "-s" can indicate possession or plurality).
- Morphological ambiguity can also arise from the way morphemes combine to form words, leading to multiple possible interpretations.
- For example, the word "unlockable" can be interpreted as "not lockable" or "able to be unlocked".

3. Productivity:

- Morphological productivity refers to the ability of a language to create new words. While some word formation processes are highly productive (e.g., adding "-ing" to verbs to create nouns), others are more restricted.
- Computational models need to be able to predict and handle the creation of new words, which can be challenging due to the varying degrees of productivity.
- This is particularly relevant in areas like natural language processing, where systems need to be able to understand and generate novel words.

Models and Techniques for morphological analysis

Finite State Transducers (FST): A common approach for morphological parsing, inputting words and outputting their stems and modifiers.

Indexed Lookup Methods: Using radix trees to analyze word forms, though can be less effective for morphologically complex languages.

Neural Networks: Increasingly used, especially for languages with abundant training data, to learn character-level language models without explicit morphological parsing.