

Consolidation Week

Weeks 1 & 2

Basic Text Processing

Regular Expressions

Regular expressions

A formal language for specifying text strings

How can we search for any of these?

- woodchuck
- woodchucks
- Woodchuck
- Woodchucks



Regular Expressions: Disjunctions

Letters inside square brackets []

Pattern	Matches
<code>[wW]oodchuck</code>	Woodchuck, woodchuck
<code>[1234567890]</code>	Any digit

Ranges `[A-Z]`

Pattern	Matches	
<code>[A-Z]</code>	An upper case letter	<u>D</u> renched Blossoms
<code>[a-z]</code>	A lower case letter	<u>m</u> y beans were impatient
<code>[0-9]</code>	A single digit	Chapter <u>1</u> : Down the Rabbit Hole

Regular Expressions: Negation in Disjunction

Negations [^Ss]

- Carat means negation only when first in []

Pattern	Matches	
[^A-Z]	Not an upper case letter	O <u>y</u> fn pripetchik
[^Ss]	Neither 'S' nor 's'	<u>I</u> have no exquisite reason"
[^e^]	Neither e nor ^	Look h <u>e</u> re
a^b	The pattern a carat b	Look up <u>a^b</u> now

Regular Expressions: More Disjunction

Woodchuck is another name for groundhog!

The pipe | for disjunction

Pattern	Matches
<code>groundhog woodchuck</code>	woodchuck
<code>yours mine</code>	yours
<code>a b c</code>	= <code>[abc]</code>
<code>[gG]roundhog [Ww]oodchuck</code>	Woodchuck



Regular Expressions: ? * + .

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



Stephen C Kleene

Kleene *, Kleene +

Regular Expressions: Anchors [^] ^{\$}

Pattern	Matches
[^] [A-Z]	<u>P</u> alo Alto
[^] [[^] A-Za-z]	<u>1</u> " <u>H</u> ello"
\. ^{\$}	The end <u>.</u>
.\sup>\$	The end <u>?</u> The end <u>!</u>

Example

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

the

Misses capitalized examples

[tT]he

Incorrectly returns other or theology

[^a-zA-Z][tT]he[^a-zA-Z]

Errors

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

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

False positives (Type I errors)

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

False negatives (Type II errors)

Errors cont.

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

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

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

Summary

Regular expressions play a surprisingly large role

- Sophisticated sequences of regular expressions are often the first model for any text processing text

For hard tasks, we use machine learning classifiers

- But regular expressions are still used for pre-processing, or as features in the classifiers
- Can be very useful in capturing generalizations

Basic Text Processing

Regular Expressions

Basic Text Processing

More Regular Expressions: Substitutions and ELIZA

Substitutions

Substitution in Python and UNIX commands:

```
s/regex1/pattern/
```

e.g.:

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

Capture Groups

- Say we want to put angles around all numbers:
the 35 boxes → *the <35> boxes*
- Use parens () 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?/

Example Multiple Choice Question

Which of the following regular expressions will correctly match email addresses in a text? Assume the email addresses follow the standard format of username@domain.extension.

A) `^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`

B) `\[a-zA-Z]+@[a-zA-Z]+\.[a-zA-Z]{2,4}$`

C) `[a-zA-Z0-9._%+-]+#[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$`

D) `^[a-zA-Z0-9._%+-]@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`

Example Multiple Choice Question

Which of the following regular expressions will correctly match email addresses in a text? Assume the email addresses follow the standard format of username@domain.extension.

A) `^[a-zA-Z0-9._%+~]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`

B) `\[a-zA-Z]+@[a-zA-Z]+\.[a-zA-Z]{2,4}$`

C) `[a-zA-Z0-9._%+~]+#[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$`

D) `^[a-zA-Z0-9._%+~]@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`

Option A is correct because it matches any string that starts with one or more characters from the set `a-zA-Z0-9._%+~` (which includes letters, digits, dots, underscores, percent signs, pluses, and hyphens), followed by an `@` symbol, then one or more characters from the set `a-zA-Z0-9.-` (which includes letters, digits, dots, and hyphens), followed by a dot `.` and two or more letters, which fits the standard format of an email address.

Option B is incorrect because it mistakenly uses square brackets around the first part of the expression, which are meant for character classes, not for grouping, and it restricts the domain extension to only 2 to 4 characters, excluding newer domain extensions that are longer than 4 characters.

Option C uses a `#` symbol instead of an `@` symbol to separate the username from the domain, which does not match the standard format of email addresses.

Option D is incorrect because it lacks the `+` after `[a-zA-Z0-9._%+~]`, which means it would only match one character from the set before the `@` symbol, and it also starts with `^` but does not include a quantifier to allow for multiple characters before the `@` symbol.

Basic Text Processing

More Regular Expressions: Substitutions and ELIZA

Basic Text Processing

Words and Corpora

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 $.67 < \beta < .75$

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 Language 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 worry ... 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

Geburu 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

Words and Corpora

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
```

Change all non-alpha to newlines

```
| sort
```

Sort in alphabetical order

```
| uniq -c
```

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-Za-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
8954 d
```

What happened here?

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

Word tokenization

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.

Example Multiple Choice Question

When applying the Byte Pair Encoding (BPE) algorithm to a large text corpus, what is a key property of the BPE tokens that makes this algorithm particularly effective for natural language processing tasks?

- A) Uniform Token Size: BPE ensures that all tokens are of the same length, facilitating fixed-size input vectors for machine learning models.
- B) Fixed Vocabulary Size: BPE generates a predefined number of tokens, regardless of the size of the corpus it is applied to.
- C) Subword Tokenization: BPE splits words into smaller, more manageable subword units, allowing the model to handle rare words, morphological variations, and out-of-vocabulary words more effectively.
- D) Lossless Compression: BPE significantly reduces the size of the text corpus by eliminating redundant characters, leading to a form of lossless compression.

BPE MCQ

When applying the Byte Pair Encoding (BPE) algorithm to a large text corpus, what is a key property of the BPE tokens that makes this algorithm particularly effective for natural language processing tasks?

A) Uniform Token Size: BPE ensures that all tokens are of the same length, facilitating fixed-size input vectors for machine learning models.

B) Fixed Vocabulary Size: BPE generates a predefined number of tokens, regardless of the size of the corpus it is applied to.

C) Subword Tokenization: BPE splits words into smaller, more manageable subword units, allowing the model to handle rare words, morphological variations, and out-of-vocabulary words more effectively.

D) Lossless Compression: BPE significantly reduces the size of the text corpus by eliminating redundant characters, leading to a form of lossless compression.

Option A (Uniform Token Size): This is incorrect because BPE does not ensure that all tokens are of the same length. The size of the tokens can vary depending on the frequency of character pairs in the corpus.

Option B (Fixed Vocabulary Size): While BPE does allow for a controlled vocabulary size, the key property that makes it effective is not the fixed size of the vocabulary but how it creates that vocabulary through subword units.

Option C (Subword Tokenization): This is correct. One of the main advantages of BPE is its ability to break down words into smaller units (subwords), which helps in dealing with rare words, morphological variations, and words not seen during training (out-of-vocabulary words). This property makes BPE particularly useful for natural language processing tasks because it improves the model's ability to understand and generate text by recognizing patterns at the subword level.

Option D (Lossless Compression): This choice is misleading. While BPE can make text representations more efficient by reducing redundancy, its primary purpose in natural language processing is not to compress text in a lossless manner but to improve the handling of a wide range of vocabulary, including rare and unknown words, through subword tokenization.

Example Multiple Choice Question

Given the corpus: "low low low low low lowest lowest newer newer newer newer newer newer wider wider wider new new", you apply the Byte Pair Encoding (BPE) algorithm to learn tokens. In the first iteration of BPE, you will identify the most frequent pair of adjacent characters to merge. Which of the following character pairs would be merged first based on their frequency in the corpus?

A) er

B) ew

C) lo

D) ne

Multiple Choice Question

Given the corpus: "low low low low low lowest lowest newer newer newer newer newer wider wider wider new new", you apply the Byte Pair Encoding (BPE) algorithm to learn tokens. In the first iteration of BPE, you will identify the most frequent pair of adjacent characters to merge. Which of the following character pairs would be merged first based on their frequency in the corpus?

A) er

B) ew

C) lo

D) ne

Option A (er): The pair er is a part of the words "newer", "wider", and "lower", appearing multiple times across the corpus. Specifically, it appears 6 times in "newer", 3 times in "wider", and is also present in "lowest", making it highly frequent.

Option B (ew): While ew is a common pair in the word "newer", it does not occur as frequently as the er pair across the entire corpus.

Option C (lo): The pair lo appears at the beginning of "low" and "lowest". Although it is frequent, especially with "low" appearing five times, it is not as common as the er pair when considering the presence of "newer" across the corpus.

Option D (ne): The pair ne appears in "newer" and "new". It is frequent but does not surpass er in terms of overall frequency within the given corpus, considering the multiple occurrences of "newer".

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

Merge **e r** to **er**

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, er

BPE

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, e r

Merge **er _** to **er_**

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, e r, e r

BPE

corpus

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

vocabulary

, d, e, i, l, n, o, r, s, t, w, er, er

Merge **n e** to **ne**

corpus

5 l o w _
2 l o w e s t _
6 ne w er_
3 w i d er_
2 ne w _

vocabulary

, d, e, i, l, n, o, r, s, t, w, er, er, ne

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*

Basic Text Processing

Byte Pair Encoding

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
- am, is, be, are

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*

Lemmatization is done by Morphological Parsing

Morphemes:

- The small meaningful units that make up words
- **Stems**: The core meaning-bearing units
- **Affixes**: Parts that adhere to stems, often with grammatical functions

Morphological Parsers:

- Parse *cats* into two morphemes *cat* and *s*
- Parse Spanish *amaren* ('if in the future they would love') into morpheme *amar* 'to love', and the morphological features *3PL* and *future subjunctive*.

Stemming

Reduce terms to stems, chopping off affixes crudely

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



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

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:

ATIONAL → ATE (e.g., relational → relate)

ING → ϵ if stem contains vowel (e.g., motoring → motor)

SSES → SS (e.g., grasses → grass)

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.

MCQ Segmentation

What is a significant challenge in sentence segmentation within Natural Language Processing (NLP), and how does it complicate the task?

A) Ambiguity in punctuation usage: The challenge lies in the fact that punctuation marks, such as periods, can serve multiple purposes (e.g., denoting abbreviations, decimals, end of sentences), making it difficult for algorithms to accurately determine sentence boundaries.

B) Lack of capitalization: Since all words can potentially start with a capital letter, distinguishing the beginning of a sentence based solely on capitalization is challenging, leading to inaccuracies in identifying sentence boundaries.

C) Uniform sentence length: The misconception that sentences must be of a uniform length leads to difficulties in segmentation, as algorithms struggle to accurately predict the end of longer or shorter sentences than the average.

D) Cross-language consistency: The challenge arises from the assumption that sentence segmentation rules are consistent across languages, which is not the case, leading to errors when applying the same segmentation model to texts in different languages.

MCQ Segmentation

What is a significant challenge in sentence segmentation within Natural Language Processing (NLP), and how does it complicate the task?

A) Ambiguity in punctuation usage: The challenge lies in the fact that punctuation marks, such as periods, can serve multiple purposes (e.g., denoting abbreviations, decimals, end of sentences), making it difficult for algorithms to accurately determine sentence boundaries.

B) Lack of capitalization: Since all words can potentially start with a capital letter, distinguishing the beginning of a sentence based solely on capitalization is challenging, leading to inaccuracies in identifying sentence boundaries.

C) Uniform sentence length: The misconception that sentences must be of a uniform length leads to difficulties in segmentation, as algorithms struggle to accurately predict the end of longer or shorter sentences than the average.

D) Cross-language consistency: The challenge arises from the assumption that sentence segmentation rules are consistent across languages, which is not the case, leading to errors when applying the same segmentation model to texts in different languages.

Option A (Ambiguity in punctuation usage): This is the correct answer. The versatility of punctuation marks, especially periods, poses a significant challenge in sentence segmentation. For instance, periods are used in abbreviations (e.g., "Dr."), decimals (e.g., "3.14"), and to mark the end of sentences. This ambiguity complicates the task for segmentation algorithms, which must discern the context to accurately identify sentence boundaries.

Option B (Lack of capitalization): This option misrepresents the issue. While capitalization is a helpful cue in sentence segmentation, especially in languages like English, the primary challenge is not the lack of capitalization but rather the correct interpretation of punctuation and contextual cues.

Option C (Uniform sentence length): Sentence length varies widely, and NLP algorithms do not assume uniformity in sentence length for segmentation. This option does not represent a real challenge in sentence segmentation.

Option D (Cross-language consistency): While it's true that sentence segmentation rules can vary across languages, leading to challenges in multilingual NLP applications, the core issue in sentence segmentation itself is distinguishing between the different uses of punctuation within the same language. Cross-language consistency is more about the adaptation of models to different linguistic rules rather than a direct challenge in the segmentation process itself.

Basic Text Processing

Word Normalization and other issues

Minimum Edit Distance

Definition of Minimum Edit
Distance

How similar are two strings?

Spell correction

- The user typed “graffe”

Which is closest?

- graf
- graft
- grail
- giraffe

- Computational Biology

- Align two sequences of nucleotides

```
AGGCTATCACCTGACCTCCAGGCCGATGCCC  
TAGCTATCACGACCGCGGTCGATTGCCCCGAC
```

- Resulting alignment:

```
-AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---  
TAG-CTATCAC--GACCGC--GGTCGATTGCCCCGAC
```

- Also for Machine Translation, Information Extraction, Speech Recognition

Edit Distance

The minimum edit distance between two strings

Is the minimum number of editing operations

- Insertion
- Deletion
- Substitution

Needed to transform one into the other

Minimum Edit Distance

Two strings and their **alignment**:

I	N	T	E	*	N	T	I	O	N
*	E	X	E	C	U	T	I	O	N

Minimum Edit Distance

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

If each operation has cost of 1

- Distance between these is 5

If substitutions cost 2 (Levenshtein)

- Distance between them is 8

Alignment in Computational Biology

Given a sequence of bases

```
AGGCTATCACCTGACCTCCAGGCCGATGCCC  
TAGCTATCACGACCGCGGGTCGATTGCCCGAC
```

An alignment:

```
-AGGCTATCACCTGACCTCCAGGCCGA--TGCCC---  
TAG-CTATCAC--GACCGC--GGTCGATTGCCCGAC
```

Given two sequences, align each letter to a letter or gap

Other uses of Edit Distance in NLP

Evaluating Machine Translation and speech recognition

R Spokesman confirms senior government adviser was appointed

H Spokesman said the senior adviser was appointed

S

I

D

I

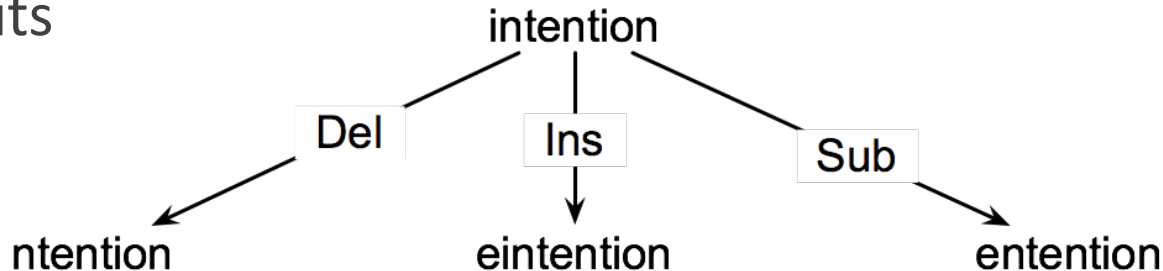
Named Entity Extraction and Entity Coreference

- IBM Inc. announced today
- IBM profits
- Stanford Professor Jennifer Eberhardt announced yesterday
- for Professor Eberhardt...

How to find the Min Edit Distance?

Searching for a path (sequence of edits) from the start string to the final string:

- **Initial state:** the word we're transforming
- **Operators:** insert, delete, substitute
- **Goal state:** the word we're trying to get to
- **Path cost:** what we want to minimize: the number of edits



Minimum Edit as Search

But the space of all edit sequences is huge!

- We can't afford to navigate naïvely
- Lots of distinct paths wind up at the same state.
 - We don't have to keep track of all of them
 - Just the shortest path to each of those revisited states.

Defining Min Edit Distance

For two strings

- X of length n
- Y of length m

We define $D(i,j)$

- the edit distance between $X[1..i]$ and $Y[1..j]$
 - i.e., the first i characters of X and the first j characters of Y
- The edit distance between X and Y is thus $D(n,m)$

MCQ Min Edit Distance

The minimum edit distance algorithm is widely used in NLP to measure the similarity between two strings. Which of the following best describes the actions counted by the minimum edit distance to transform one string into another?

- A) Counting the number of words that need to be added, deleted, or replaced in a text to match another text.
- B) Calculating the number of characters that differ between two texts, without considering the order of characters.
- C) Determining the minimum number of character insertions, deletions, and substitutions required to change one string into another.
- D) Measuring the total number of characters in two strings, then finding the difference to establish a distance metric.

MCQ MED

The minimum edit distance algorithm is widely used in NLP to measure the similarity between two strings. Which of the following best describes the actions counted by the minimum edit distance to transform one string into another?

A) Counting the number of words that need to be added, deleted, or replaced in a text to match another text.

B) Calculating the number of characters that differ between two texts, without considering the order of characters.

C) Determining the minimum number of character insertions, deletions, and substitutions required to change one string into another.

D) Measuring the total number of characters in two strings, then finding the difference to establish a distance metric.

Option A suggests a word-level operation which, although relevant in some NLP tasks, does not accurately describe the minimum edit distance which operates at the character level.

Option B inaccurately portrays minimum edit distance by suggesting it calculates character differences without regard to order, which is not true. The algorithm considers the sequence of characters and their positions.

Option C is correct because the minimum edit distance algorithm indeed calculates the least number of insertions, deletions, and substitutions needed to transform one string into another. This metric is crucial for applications like spell checking, plagiarism detection, and more.

Option D misrepresents the concept by implying that the algorithm simply measures the total number of characters in two strings and compares them, which overlooks the nuanced operations (insertions, deletions, substitutions) that the minimum edit distance actually involves.

Minimum Edit Distance

Definition of Minimum Edit Distance

Minimum Edit Distance

Computing Minimum Edit Distance

Dynamic Programming for Minimum Edit Distance

Dynamic programming: A tabular computation of $D(n,m)$
Solving problems by combining solutions to subproblems.

Bottom-up

- We compute $D(i,j)$ for small i,j
- And compute larger $D(i,j)$ based on previously computed smaller values
- i.e., compute $D(i,j)$ for all i ($0 < i < n$) and j ($0 < j < m$)

Defining Min Edit Distance (Levenshtein)

Initialization

$$D(i, 0) = i$$

$$D(0, j) = j$$

Recurrence Relation:

For each $i = 1 \dots N$

For each $j = 1 \dots M$

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

Termination:

$D(N, M)$ is distance


The Edit Distance Table

N	9									
O	8									
I	7									
T	6									
N	5									
E	4									
T	3									
N	2									
I	1									
#	0	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

The Edit Distance Table

N	9									
O	8									
I	7									
T	6									
N	5									
E	4									
T	3									
N	2									
I	1									
#	0	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases} \end{cases}$$



Edit Distance

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases} \end{cases}$$

N	9									
O	8									
I	7									
T	6									
N	5									
E	4									
T	3									
N	2									
I	1									
#	0	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

The Edit Distance Table

N	9	8	9	10	11	12	11	10	9	8
O	8	7	8	9	10	11	10	9	8	9
I	7	6	7	8	9	10	9	8	9	10
T	6	5	6	7	8	9	8	9	10	11
N	5	4	5	6	7	8	9	10	11	10
E	4	3	4	5	6	7	8	9	10	9
T	3	4	5	6	7	8	7	8	9	8
N	2	3	4	5	6	7	8	7	8	7
I	1	2	3	4	5	6	7	6	7	8
#	0	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

Minimum Edit Distance

Computing Minimum Edit Distance

Minimum Edit Distance

Backtrace for Computing Alignments

Computing alignments

Edit distance isn't sufficient

- We often need to **align** each character of the two strings to each other

We do this by keeping a “backtrace”

Every time we enter a cell, remember where we came from

When we reach the end,

- Trace back the path from the upper right corner to read off the alignment

Edit Distance

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \begin{cases} 2; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases} \end{cases}$$

N	9									
O	8									
I	7									
T	6									
N	5									
E	4									
T	3									
N	2									
I	1									
#	0	1	2	3	4	5	6	7	8	9
	#	E	X	E	C	U	T	I	O	N

MinEdit with Backtrace

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

Adding Backtrace to Minimum Edit Distance

Base conditions:

$$D(i, 0) = i$$

$$D(0, j) = j$$

Termination:

$$D(N, M) \text{ is distance}$$

Recurrence Relation:

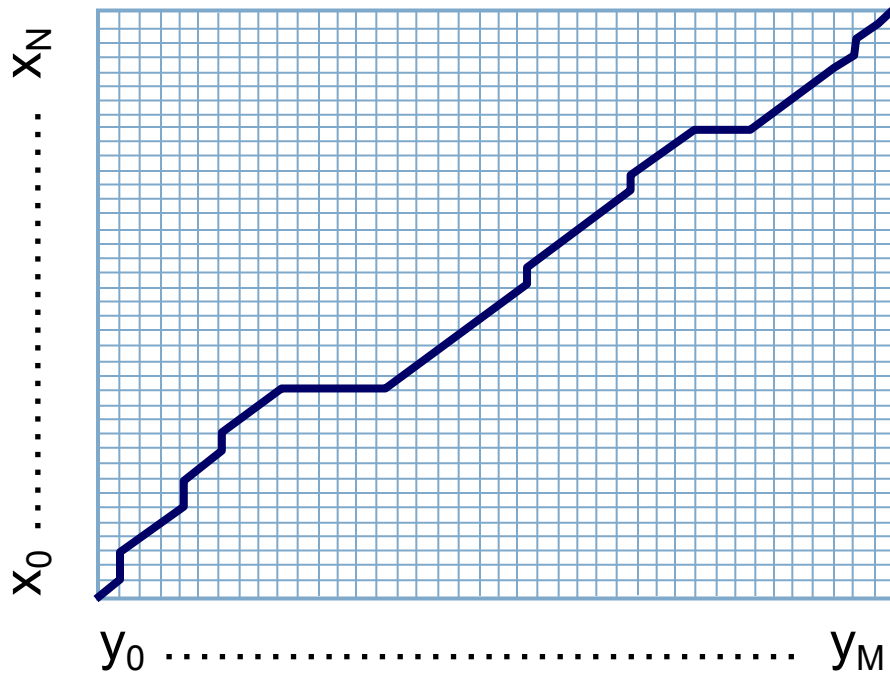
For each $i = 1 \dots M$

For each $j = 1 \dots N$

$$D(i, j) = \min \begin{cases} D(i-1, j) + 1 & \text{deletion} \\ D(i, j-1) + 1 & \text{insertion} \\ D(i-1, j-1) + \begin{cases} 2; & \text{if } X(i) \neq Y(j) \\ 0; & \text{if } X(i) = Y(j) \end{cases} & \text{substitution} \end{cases}$$

$$\text{ptr}(i, j) = \begin{cases} \text{LEFT} & \text{insertion} \\ \text{DOWN} & \text{deletion} \\ \text{DIAG} & \text{substitution} \end{cases}$$

The Distance Matrix



Every non-decreasing path
from $(0,0)$ to (M, N)

corresponds to
an alignment
of the two sequences

An optimal alignment is composed
of optimal subalignments

Result of Backtrace

Two strings and their **alignment**:

I	N	T	E	*	N	T	I	O	N
*	E	X	E	C	U	T	I	O	N

Performance

Time:

$O(nm)$

Space:

$O(nm)$

Backtrace

$O(n+m)$

Minimum Edit Distance

Backtrace for Computing Alignments

Minimum
Edit
Distance

Weighted Minimum Edit
Distance

Weighted Edit Distance

Why would we add weights to the computation?

- Spell Correction: some letters are more likely to be mistyped than others
- Biology: certain kinds of deletions or insertions are more likely than others

Confusion matrix for spelling errors

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	0	2	43	0	0	4	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



Weighted Min Edit Distance

Initialization:

$$D(0,0) = 0$$

$$D(i,0) = D(i-1,0) + \text{del}[x(i)]; \quad 1 < i \leq N$$

$$D(0,j) = D(0,j-1) + \text{ins}[y(j)]; \quad 1 < j \leq M$$

Recurrence Relation:

$$D(i,j) = \min \begin{cases} D(i-1,j) + \text{del}[x(i)] \\ D(i,j-1) + \text{ins}[y(j)] \\ D(i-1,j-1) + \text{sub}[x(i),y(j)] \end{cases}$$

Termination:

$D(N,M)$ is distance

Where did the name, dynamic programming, come from?

...The 1950s were not good years for mathematical research. [the] Secretary of Defense ...had a pathological fear and hatred of the word, research...

I decided therefore to use the word, “**programming**”.

I wanted to get across the idea that this was dynamic, this was multistage... I thought, let's ... take a word that has an absolutely precise meaning, namely **dynamic**... it's impossible to use the word, **dynamic**, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible.

Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to.”

Richard Bellman, “Eye of the Hurricane: an autobiography” 1984.

Minimum
Edit
Distance

Weighted Minimum Edit
Distance