

LINGUISTICS COURSE

Computational Linguistics

How machines count, tag, parse and measure human language

10 Chapters | Philip Osztromok | Generated with Claude

COMPUTATIONAL LINGUISTICS

Contents

Chapter 1 Language as Data 2

Chapter 2 Words and Tokens 8

Chapter 3 Corpora 14

Chapter 4 Morphology by Machine 20

Chapter 5 Language Models by Counting 27

Chapter 6 Grammars and Parsing 33

Chapter 7 Tagging and Sequences 40

Chapter 8 Meaning 48

Chapter 9 Measuring Success 56

Chapter 10 Limits and Society 62

Computational Linguistics

Course 1 · Chapter 1 · Language as Data

A computer does not see words. It sees numbers. Before a program can count a word, parse a sentence or translate a paragraph, someone has to decide how language becomes numbers, and that decision already contains linguistics. This chapter says what the field is, where it came from, and then looks at the very first step: how text is stored.

> **THE IDEA IN A BOX**
>
> Wikipedia describes computational linguistics as an interdisciplinary field "concerned with the computational modelling of natural language". In this course, the question behind every chapter is the same: *what exactly is the program being asked to do with language, and how would we know it did it?*

What the Field Is

- **Definition.** The Computational linguistics article calls it an interdisciplinary area concerned with the computational modelling of natural language and with finding suitable computational approaches to linguistic questions. It is closely related to mathematical linguistics.
- **Origins.** It overlapped with AI from the 1950s US efforts to use computers for machine translation, especially of Russian scientific journals into English. Researchers expected rule-based methods could also capture lexicon, morphology, syntax and semantics. After those approaches failed, David Hays coined the term to separate the field from AI, and helped found the ACL and ICCL in the 1970s and 1980s.
- **Neighbours.** The page lists linguistics, computer science, AI, mathematics, logic, philosophy, cognitive science, psycholinguistics, anthropology and neuroscience. It says the translation effort grew into the wider field of natural language processing (NLP).

Computational linguistics and NLP

The two pages do not give a definitional distinction. The NLP article defines NLP as "the processing of natural language information by a computer" and a subfield of computer science closely associated with AI, and says it "is also related to" computational linguistics. The Computational linguistics article implies NLP is the broader field that grew out of its translation

origins. In practice this course uses the split described in its outline: NLP 1 teaches the machine-learning pipeline; this course keeps the linguist's questions in view. That working split is mine, not the sources'.

A short timeline (from the NLP article)

DATE	EVENT OR PERIOD
1950	Alan Turing published "Computing Machinery and Intelligence", proposing what is now the Turing test
1954	The Georgetown experiment demonstrated automatic translation of more than sixty Russian sentences into English; its authors predicted translation would be solved within a few years, but progress was much slower
1950s to early 1990s	Symbolic NLP: hand-written rules; the 1980s and early 1990s are called its heyday
1990s onward	Statistical NLP: machine learning replaced hand-written rules starting in the late 1980s; in 2003 a neural probabilistic language model outperformed the word n-gram model
From 2015	Neural methods increasingly replace traditional statistical approaches

The Tasks, Mapped to This Course

The NLP article lists common tasks. The grouping below is the article's; which chapter of this course covers each group is my own mapping.

LEVEL	TASKS THE ARTICLE LISTS	CHAPTER HERE
Text and speech	tokenization, optical character recognition, speech recognition, text-to-speech	1 and 2 (text); 10 (speech)
Morphology	lemmatization, morphological segmentation, part-of-speech tagging, stemming	4, 7
Syntax	parsing, sentence breaking, grammar induction	6
Semantics	named entity recognition, word-sense disambiguation, sentiment analysis, semantic role labelling	8
Applications	machine translation, question answering, summarization	10

Text Is Numbers: Unicode

The Unicode article calls Unicode "a character encoding standard maintained by the Unicode Consortium" that aims to support text in the world's writing systems that can be digitized. It states that version 18.0 defines 172,808 characters and 175 scripts (a figure I have not checked against the standard itself).

- **Code points.** Unicode gives each character a number, a code point, in a codespace from 0 to 1114111, written U+0000 to U+10FFFF: "U+" plus hexadecimal, with at least four digits (the article's example is U+00F7, DIVISION SIGN).
- **Bytes.** A code point is an abstract number. Text is stored by an encoding that maps code points to sequences of bytes.
- **UTF-8.** The most widely used encoding by a large margin, partly because of its backwards-compatibility with ASCII. In UTF-8, code points in the Basic Multilingual Plane take one to three bytes and supplementary-plane code points take four. UTF-16 uses a single code unit for the Basic Multilingual Plane and surrogate pairs beyond it.
- **Combining characters.** A combining mark can be added after a base character, so "é" can be one precomposed code point or "e" plus a combining acute accent; the article says canonical equivalence makes these interchangeable in practice.
- **Before Unicode.** Many encodings were built for single contexts and were often incompatible, so text in one could appear as garbage when read with another (the article calls one such case mojibake).

Seeing it in Python

The examples below are my own; I ran each and copied its output. Hungarian is used because it has accented letters beyond the ASCII range.

```
for ch in ["A", "é", "ő", "語"]:
    print(ch, "U+%04X" % ord(ch), len(ch.encode("utf-8")), "bytes in UTF-8")
```

```
A U+0041 1 bytes in UTF-8
é U+00E9 2 bytes in UTF-8
ő U+0151 2 bytes in UTF-8
語 U+8A9E 3 bytes in UTF-8
```

So one character is not one byte. Counting "length" depends on what you count:

```
s = "Szia, világ!"
print(len(s), len(s.encode("utf-8")))
```

```
12 13
```

Twelve characters, thirteen bytes: the "á" takes two. And the same visible word can be two different strings:

```
import unicodedata
a = "café"      # "é" as one code point
b = "café"     # "e" plus a combining acute accent
print(a, b, a == b, len(a), len(b))
print(unicodedata.normalize("NFC", b) == a)
```

```
café café False 4 5
True
```

The two strings look the same on screen but are not equal until they are normalised to a single form (NFC joins the pieces). A program that counts words without normalising would count "café" twice.

Finally, the garbage the article describes. Here a UTF-8 string is read back with the wrong encoding:

```
print("café".encode("utf-8").decode("latin-1"))
```

```
cafÃ©
```

WATCH OUT: THIS IS NOT THE ARTICLE'S MOJIBAKE EXAMPLE

The Unicode article describes mojibake through C1 control code points misdecoded as Windows-1252. My example above is a different mix-up (UTF-8 bytes read as Latin-1) that produces the same kind of damage. I give it as my own illustration, not as the article's case.

NOTE ON SOURCES

This chapter uses Wikipedia's Computational linguistics (read in full), Natural language processing (read to 100,000 of 142,965 characters; it ends partway through the references) and Unicode (read to

100,000 of 257,769 characters; it ends partway through "Standardized subsets", so later sections were not read). The Computational linguistics page does not mention the Cold War, and the Unicode page gives no UTF-32 detail and does not use U+0041 as an example, so neither appears as sourced. All Python examples and their outputs are mine, run on this machine with `py -X utf8`; the Unicode figure for version 18.0 is the article's own and unchecked. The task-to-chapter mapping is mine.

Try It Yourself

EXERCISE 1

In Python, print the code point (in U+XXXX form) and the number of UTF-8 bytes for "n", "ñ", "€" and "😀". What pattern do you see between the size of the code point and the number of bytes?

View solution

EXERCISE 2

The word "naïve" has 5 characters. Use `unicodedata.normalize("NFD", ...)` to split the "ï" into pieces. How many characters does the result have, and what are the pieces called?

View solution

EXERCISE 3

Sort these tasks by level using the NLP article's grouping: stemming, parsing, tokenization, word-sense disambiguation, machine translation. Then say in two sentences why this course treats the history of the Georgetown experiment as a warning and not a success story.

View solution

CHAPTER 1 QUICK REFERENCE

- Computational linguistics: computational modelling of natural language; NLP grew from its machine-translation origins
- Eras per the NLP article: symbolic (hand-written rules), statistical, neural (increasingly from 2015)

- Unicode: code points U+0000 to U+10FFFF; encodings (UTF-8 most used) turn them into bytes
- Characters are not bytes, and identical-looking strings may differ until normalised

COMING NEXT

Computational Linguistics 2 asks the question that sounds easy: what counts as a word?

Computational Linguistics

Course 1 · Chapter 2 · Words and Tokens

Chapter 1 turned text into characters and bytes. The next question sounds trivial: where do the words begin and end? Almost every program that handles language starts by cutting text into pieces, and the way it cuts decides everything it counts afterwards.

> **THE IDEA IN A BOX**
>
> A **token** is a piece the program chooses to treat as a unit. It is often a word, but "word" has no single definition, so a tokenizer is always a set of decisions, and those decisions can be right or wrong for the job.

What Is a Word?

Wikipedia's Word article says "there is no consensus among linguists on its definition", and that different criteria apply at different levels of analysis without converging. It names three of them:

KIND	DEFINED BY	THE ARTICLE'S WORDING
Orthographic word	writing	"a segment indicated by blank spaces in writing or print"
Phonological word	sound	"a unit larger or equal to a syllable", found through segmental or prosodic features or phonological rules
Grammatical word	morphology and inflection	distinct from word-forms

- **Lexeme and word form.** A lexeme is the dictionary-listed unit and covers all its inflected forms, so *teapot* and *teapots* belong to one lexeme. Chapter 4 returns to this.
- **No spaces.** Chinese, Japanese, Thai and Lao often do not separate words with spaces. The article says Mandarin is analytic, so delimiting words orthographically is "unnecessary", and that Japanese uses kanji/kana switching as a soft cue.
- **Compounds and contractions.** Compound words such as *cowboys* and *typewriter* have more than one root; contractions like *can't* are described as words formed from multiple words.

Tokenization

The Lexical analysis article is mostly about programming languages, but it has a passage on natural language. It says a machine does not implicitly segment text on spaces the way a human reader does, so delimiters must be defined explicitly, and defining a "word" is hard. Its edge cases are contractions, hyphenated words, emoticons and URIs; its example is *New York-based*, where a naive tokenizer may split at the space when the hyphen is the better break. It adds that tokenization is especially difficult for languages written in *scriptio continua*, naming Ancient Greek, Chinese and Thai, and that agglutinative languages such as Korean also complicate it. Suggested remedies are more complex heuristics, lookup tables of special cases, and language models that identify collocations in a later step.

Three tokenizers, one sentence

The sentence and code below are my own. I ran each and copied the output.

```
import re

s = "Don't call the New York-based e-mail team before 5 p.m., said Dr. Smith."
print(s.split())
print(re.findall(r"\w+", s))
print(re.findall(r"\w+(?:[-']\w+)*", s))
```

```
["Don't", 'call', 'the', 'New', 'York-based', 'e-mail', 'team', 'before', '5', 'p.m.,',
['Don', 't', 'call', 'the', 'New', 'York', 'based', 'e', 'mail', 'team', 'before', '5',
["Don't", 'call', 'the', 'New', 'York-based', 'e-mail', 'team', 'before', '5', 'p', 'm',
```

TOKENIZER	WHAT IT GETS RIGHT	WHAT IT GETS WRONG
Split on spaces	keeps *Don't*, *York-based*, *e-mail* whole	leaves punctuation stuck on *p.m.*, and *Smith.*
Letters and digits only	removes the punctuation	tears *Don't* into *Don* + *t*, and breaks *e-mail* and *p.m.*
Letters, digits, inner hyphens and apostrophes	keeps *Don't*, *York-based*, *e-mail*	still breaks *p.m.* into *p* and *m*, and loses the full stop in *Dr.*

None of the three is correct everywhere. Whether *New York-based* should be one token, two or three depends on what you want to do next, which is the Lexical analysis article's point.

When there are no spaces

```
zh = "我喜欢咖啡"
print(zh.split())
print(len(zh))
```

```
['我喜欢咖啡']
5
```

Splitting on spaces returns the whole string as a single "word", because nothing in it is a space. Cutting it into words needs knowledge of the language, not just a rule about spaces. I have given the string only to show that failure; I do not segment it here, and I do not claim a segmentation, because I have not checked one against a source.

Types and Tokens

The Type–token distinction article credits Charles Sanders Peirce, who drew the distinction in 1906. Types are "abstract descriptive concepts"; tokens are "objects that instantiate concepts", the particular occurrences of a type. Because each type can have many tokens, there are generally more tokens than types. Its example is "A rose is a rose is a rose": three word types, with three tokens of *a*, two of *is* and three of *rose*.

```
from collections import Counter

text = "A rose is a rose is a rose."
words = text.lower().replace(".", "").split()
print(len(words), len(set(words)))
print(Counter(words))
```

```
8 3
Counter({'a': 3, 'rose': 3, 'is': 2})
```

Eight tokens, three types, matching the article's counts. The article does not mention the type-token ratio, so this course does not rely on it here.

Word Frequencies and Zipf's Law

The Zipf's law article states that in a natural-language text, a word's frequency is roughly inversely proportional to its rank: "the most common word occurs approximately twice as often as the next common one" (and about three times as often as the third). Its Brown Corpus example: *the* is the most frequent word at nearly 7% of occurrences (69,971 of slightly over 1 million), *of* is second at slightly over 3.5% (36,411) and *and* follows with 28,852.

RANK	WORD	COUNT (ARTICLE)	ZIPF PREDICTION FROM RANK 1 (MY ARITHMETIC)
1	the	69,971	69,971
2	of	36,411	34,986
3	and	28,852	23,324

The prediction column is 69,971 divided by the rank, which I calculated: the real rank-2 count is close to half of rank 1 (ratio 1.92), but rank 3 is about 2.43 times smaller, not 3. That is the article's own caveat: the law holds only approximately.

- **History.** Named after George Kingsley Zipf, who observed the word-frequency relation in 1932 but "never claimed to have originated it"; earlier discoveries are credited to Jean-Baptiste Estoup (1916), Godfrey Dewey (1923) and Edward Condon (1928).
- **Form.** The Zipf–Mandelbrot version is frequency $\propto 1 / (\text{rank} + b)^a$, with $a \approx 1$ and $b \approx 2.7$ as fitted parameters.
- **Where it bends.** Real plots deviate at the high- and low-frequency ends, depending on language, topic, author, translation and spelling. Across a large set of English texts only about 15% fit well, rising to nearly 50% with small changes to the definition. Some languages deviate more, including Romance languages' most frequent words and East Asian languages where each morpheme is one syllable.
- **Why.** The article says the reason is still poorly understood; proposals include random-text models, the principle of least effort, and preferential attachment.
- **Uses.** It applies to most natural languages and some constructed ones, such as Esperanto and Toki Pona, and is used in authorship attribution and in analyzing the Voynich manuscript.

Why a small text will not show it

```
t = ("the cat sat on the mat and the dog sat on the log "
     "while the bird sang and the cat slept")
c = Counter(t.split())
print(len(t.split()), len(c))
for rank, (w, n) in enumerate(c.most_common(5), 1):
    print(rank, w, n)
```

```
21 12
1 the 6
2 cat 2
3 sat 2
4 on 2
5 and 2
```

Twenty-one tokens and twelve types: *the* has six, then four words tie on two. A tiny text gives ties and jumps, not the smooth curve. The law is a statement about large collections, so this chapter does not claim my toy sentence demonstrates it.

WATCH OUT: A COUNT IS NOT A MEANING

The is the commonest word because it is a grammatical word, not because it is the most important. Frequency lists answer "how often", not "how much it matters". Chapter 3 shows what a corpus can and cannot tell you.

NOTE ON SOURCES

This chapter uses Wikipedia's Word, Type–token distinction and Lexical analysis (all read in full) and Zipf's law (read to 100,000 of 121,078 characters, so its last part was not read). The Word page does not cover clitics, the Type–token page does not mention type-token ratio or corpus linguistics, and the Zipf page gives no percentage for *and*. The sentences, code, outputs and the prediction column are mine; the Chinese string is shown only to display a failure, not segmented. Reading Chapter 1's "tokenization" as the same step is my own link.

Try It Yourself

EXERCISE 1

Run the three tokenizers on "*It's a well-known fact, isn't it?*" and compare the outputs. For each of *It's*, *well-known* and *isn't*, say which tokenizer keeps it whole.

View solution

EXERCISE 2

Count the tokens and types in "*to be or not to be*", then say which is the type and which are the tokens for the word *be*. What happens to the type count if the first word is written *To* and you do not lowercase?

View solution

EXERCISE 3

Using the article's Brown Corpus counts, calculate what Zipf's law predicts for ranks 2 and 3 from rank 1, compare with the real counts, and say what that shows about calling it a "law".

View solution

CHAPTER 2 QUICK REFERENCE

- No agreed definition of "word": orthographic, phonological, grammatical
- Tokenization is a set of choices; edge cases include contractions, hyphens, abbreviations and scripts without spaces
- Type = distinct form; token = each occurrence (8 tokens, 3 types in "A rose is a rose is a rose")
- Zipf's law: frequency roughly proportional to 1 / rank, approximate and bending at the ends

COMING NEXT

Computational Linguistics 3 looks at corpora: what a body of text can show, and what it cannot.

Computational Linguistics

Course 1 · Chapter 3 · Corpora

Chapter 2 counted words in a single sentence. A real study counts them in millions. The collection a linguist or a program learns from is called a corpus, and what it contains decides what any result built on it can possibly say.

> **THE IDEA IN A BOX**
>
> A corpus is evidence, and every body of evidence has a shape. Before trusting a count, ask **what text went in, and who or what chose it**.

What a Corpus Is

- **Definition.** Wikipedia's Corpus linguistics article calls the field "an empirical method for the study of language by text corpus". Corpora are balanced, stratified collections of real-world speech or writing, and today they are generally machine-readable.
- **Text corpus.** The Text corpus article defines one as "a dataset, consisting of natively digital and older, digitalized, language resources", which may be annotated or unannotated. It calls corpora "the main knowledge base in corpus linguistics" and says annotated ones support statistical hypothesis testing and checking how linguistic rules apply. Other uses it lists: language technology, machine translation, language teaching and philology.

Two famous corpora

CORPUS	WHAT THE CORPUS LINGUISTICS ARTICLE SAYS
Brown Corpus	a balanced corpus of about one million words of American English from 1961, analyzed by Kučera and Francis in 1967; "the first computerized corpus designed for linguistic research"
British National Corpus	"a 100 million word collection of a range of spoken and written texts", created in the 1990s by a consortium that included Oxford, Lancaster and the British Library

The article adds that Quirk's corpus "was the first modern corpus to be built with the purpose of representing the whole language", and that corpora fed dictionaries and grammars such as the

American Heritage Dictionary (1969) and Quirk et al.'s 1985 grammar.

Annotation

A raw corpus is just text. Annotation adds information. The Corpus linguistics article says it may include "structural markup, part-of-speech tagging, parsing, and numerous other representations", and that most lexical corpora are part-of-speech tagged. It also reports a disagreement: John Sinclair preferred minimal annotation, the Survey of English Usage team supported more. Wallis and Nelson's 3A framework describes the work as annotation (applying a scheme), abstraction (mapping the scheme onto a theoretical model) and analysis ("statistically probing, manipulating and generalising from the dataset").

KIND OF ANNOTATION	WHAT THE TEXT CORPUS ARTICLE SAYS
Part-of-speech tagging	adds a tag for each word's part of speech, such as verb, noun or adjective
Lemmatization	marks each word's base form
Interlinear glossing	makes annotation bilingual when the corpus language is not the researchers' working language
Treebanks	fully parsed corpora, usually smaller, "containing around one to three million words"; consistent annotation of a whole corpus is difficult

The two sources differ a little on size: Chapter 1's Computational linguistics article gives the Penn Treebank as "over 4.5 million words" of American English, above the Text corpus article's "around one to three million". I report both and do not try to reconcile them.

Looking Inside: Concordance Lines

The Key Word in Context article calls KWIC "the most common format for concordance lines": each line places a keyword in a fixed, central position with the words before and after it. The term was coined by Hans Peter Luhn; the underlying "keyword in titles" idea was first proposed for Manchester libraries in 1864 by Andrea Crestadoro.

The text, code and outputs below are mine: an invented 66-word paragraph, run through a small script.

```
import re
from collections import Counter

text = ("The old lighthouse stood on the cliff. The keeper climbed the stairs every nigh
        "and lit the lamp. Strong tea warmed the keeper after the climb. "
        "On stormy nights the keeper drank strong tea and watched the sea. "
        "The sea was grey. The lamp was bright. The keeper made a decision: "
        "he would stay another winter. Strong winds shook the lighthouse, "
        "but the lamp never failed.")
tokens = re.findall(r"[a-z]+", text.lower())
print(len(tokens), len(set(tokens)))

def kwic(tokens, word, width=3):
    for i, t in enumerate(tokens):
        if t == word:
            left = " ".join(tokens[max(0, i - width):i]).rjust(22)
            right = " ".join(tokens[i + 1:i + 1 + width])
            print(left, "[" + t + "]", right)

kwic(tokens, "keeper")
```

```
66 40
         the cliff the [keeper] climbed the stairs
        tea warmed the [keeper] after the climb
     stormy nights the [keeper] drank strong tea
        was bright the [keeper] made a decision
```

Four lines, one keyword in the middle. A reader sees at once what tends to follow *keeper*: a verb. A caution about my own script: it throws away punctuation, so a window can cross a sentence boundary. The line "was bright the [keeper]" runs from one sentence into the next.

Collocations

The Collocation article defines a collocation, in corpus linguistics, as "a series of words or terms that co-occur more often than would be expected by chance". Its examples are *crystal clear*, *middle management*, *nuclear family* and *cosmetic surgery*, plus *make* + *decision* as a verb–object

pair. A collocation's meaning can be understood from its words; for an idiom "the meaning of the whole cannot be inferred from its parts". Statistically, association measures test whether co-occurrence is due to chance, and because most collocations prove significant, the scores mainly rank results. The page names mutual information, t-scores and log-likelihood, and notes KWIC analysis as a way to find them.

The idea in miniature, in my own code: compare how often two words appear next to each other with how often they would if the words were scattered at random. The "expected" figure here is my simplified calculation, not one of the measures the article names.

```
n = len(tokens)
c = Counter(tokens)
pairs = Counter(zip(tokens, tokens[1:]))
for pair in [("strong", "tea"), ("the", "keeper"), ("the", "lamp")]:
    obs = pairs[pair]
    exp = c[pair[0]] * c[pair[1]] / n
    print(pair, obs, round(exp, 2))
```

```
('strong', 'tea') 2 0.09
('the', 'keeper') 4 0.85
('the', 'lamp') 3 0.64
```

Strong tea occurs twice where chance predicts under 0.1; *the keeper* four times where chance predicts under 1. The ratio ranks *strong tea* as the stronger pairing. But *the keeper* also beats chance, which fits the article's remark that most collocations prove significant and scores mostly rank. A 66-word text is far too small for any of this to count as evidence about English; the point is to see the arithmetic.

What a Corpus Cannot Tell You

- **It shows the language that went in.** The Brown Corpus is American English from 1961; the BNC dates from the 1990s. Neither tells you how people write today.
- **Balance is a design choice.** The Corpus linguistics article describes corpora as "balanced, often stratified collections of authentic" texts meant to represent a variety. Someone decided what counted as balanced, and the page does not describe how.
- **Absence is not impossibility.** A construction missing from a million words may be rare, or may belong to a kind of text the corpus left out.

- **Annotation has errors and theories in it.** A tag is a decision; the Text corpus article says consistent annotation of a whole corpus is difficult.

WATCH OUT: A COUNT FROM ONE CORPUS IS NOT A FACT ABOUT THE LANGUAGE

"The word occurs 4 times per million" is a fact about *that corpus*. Whether it generalises depends on how the texts were chosen. The sources I read give corpus sizes and dates, but the Corpus linguistics page says nothing about how representativeness is checked, and the Text corpus page does not cover sampling, so I make no claim about it beyond those cautions.

NOTE ON SOURCES

This chapter uses Wikipedia's Corpus linguistics, Text corpus, Collocation and Key Word in Context articles (all read in full). The Corpus linguistics page does not discuss concordance, collocation or frequency methods, corpus-based versus corpus-driven work, or criticism, and does not mention Chomsky. The Text corpus page does not mention the Brown Corpus, gives no BNC size, and does not cover sampling. The Collocation page has no Firth quotation, no "strong tea" example and no mention of pointwise mutual information, so "strong tea" is my example. The paragraph, script, outputs and the "expected" calculation are mine. Chapter 1's Penn Treebank figure comes from the Computational linguistics article.

Try It Yourself

EXERCISE 1

Using the chapter's `tokens` and `kwic`, print concordance lines for *lamp* with a window of 2. What do you notice about the first line, and what does it tell you about the script?

View solution

EXERCISE 2

Compute observed and expected counts for *the sea*, *the lighthouse* and *strong winds*. Which ratio is largest, and why should you hesitate to call it a collocation?

View solution

EXERCISE 3

You want to study how people talk about money today. Explain why the Brown Corpus and the British National Corpus, as the sources describe them, would each be a poor choice on its own, and say what kind of corpus you would want instead.

View solution

CHAPTER 3 QUICK REFERENCE

- Corpus: a balanced, stratified collection of real language, usually machine-readable
- Annotation: markup, part-of-speech tags, lemmas, parse trees (treebanks)
- KWIC: keyword centred with context on both sides
- Collocation: co-occurrence more often than chance; scores mostly rank
- A corpus shows what went in: date, variety, selection

COMING NEXT

Computational Linguistics 4 looks at words from the inside: stems, lemmas and patterns of endings.

Computational Linguistics

Course 1 · Chapter 4 · Morphology by Machine

Chapter 2 asked what counts as a word. This chapter asks a harder question: when are *walk*, *walks*, *walked* and *walking* the same word? A search for one should probably find the others, and a frequency count should probably group them. The tools for that sit between simple string handling and full linguistic analysis.

> **THE IDEA IN A BOX**
>
> There are two answers. **Stemming** chops the ends off words by rule and is fast and crude; **lemmatization** finds the dictionary form and needs more knowledge. Neither is "right"; each trades accuracy for effort.

Stemming

Wikipedia's Stemming article defines it as "the process of reducing inflected (or sometimes derived) words to their word stem, base or root form". A stem can be a fragment, not a dictionary word: the Porter algorithm reduces *argue*, *argued*, *argues*, *arguing* and *argus* to *argu*.

ALGORITHM TYPE	WHAT THE ARTICLE SAYS
Lookup table	quick and good with irregular forms, but every inflected form must be listed
Suffix stripping	rules such as removing "ed", "ing" or "ly"; easier to maintain but crude with irregular pairs like "ran" and "run"
Lemmatisation	first identifies a word's part of speech, then applies normalization rules for that category
Stochastic	probability models trained on pairs of roots and inflected forms
Hybrid	combinations, such as a small exception list backed by suffix stripping

- **The Porter stemmer.** Martin Porter published it in the July 1980 issue of *Program*; it became "the de facto standard algorithm used for English stemming". Around 2000 he released an official free implementation and later built Snowball, a framework for writing stemmers.

- **Over-stemming.** The article says Porter reduces *universal*, *university* and *universe* to *univers*: related by etymology but now very different in meaning, so conflating them can hurt search relevance.
- **Under-stemming.** Porter reduces *alumna* and *alumnae* to *alumna* but leaves *alumnus* as *alumnu* and *alumni* as *alumni*, so these are not grouped.
- **Hard languages.** Hebrew and Arabic are described as still difficult: Hebrew has nonconcatenative morphology, a writing system without vowels, and needs prefix stripping. Highly inflected languages like Turkish can have hundreds of inflected forms per root.
- **In search.** Early results for English were limited, which led some researchers to consider stemming largely irrelevant; stemming may help more in other languages. Google adopted word stemming in 2003; before that, a search for "fish" would not return "fishing".

A toy suffix stripper

The Python below is my own, not the Porter algorithm, and I could not run Porter here (no library for it was installed), so I do not compare outputs. I ran this code and copied its output.

```
def strip_suffix(word):
    for suffix in ("ing", "ed", "es", "s"):
        if word.endswith(suffix) and len(word) - len(suffix) >= 3:
            return word[:-len(suffix)]
    return word

for w in ["walked", "walking", "walks", "walk", "argued", "arguing",
          "running", "ran", "studies", "bus", "this"]:
    print(w, "->", strip_suffix(w))
```

```
walked -> walk
walking -> walk
walks -> walk
walk -> walk
argued -> argu
arguing -> argu
running -> runn
ran -> ran
studies -> studi
bus -> bus
this -> thi
```

The *walk* family collapses as hoped. But *running* becomes *runn* (not a word, and different from the stem of *runs*), *this* becomes *thi* because it ends in *s*, *studies* becomes *studi*, and *ran* is left alone, exactly the "crude with irregular pairs" problem. The length check (`>= 3`) keeps *bus* intact, but it is a patch, and it will break something else.

Lemmatization

The Lemmatization article defines it as grouping a word's inflected forms so they can be analysed as one item, identified by the lemma, the base or dictionary form: *walk*, *walking*, *walked* and *walks* share the lemma *walk*. Computationally it determines the lemma from the intended meaning, which requires identifying the part of speech and the context. The page's examples: *better* has *good* as its lemma, a link stemming misses because it requires a dictionary lookup; *meeting* may be a noun or a verb form and lemmatization picks by context.

	STEMMING	LEMMATIZATION
Works on	the word alone, without context	the word with its part of speech and context
Output	may not be a real word ("lazy" becomes "lazi")	a dictionary form
Cost	simpler and faster	depends on correct part-of-speech identification; efficient algorithms remain an open research area
In search	can improve recall but lowers precision	"a trivial way" is simple dictionary lookup

The page also says languages with long compound words need rule-based systems, which can be hand-crafted or learned from annotated corpora. A lookup lemmatizer in miniature (my code, tiny dictionary):

```
lemma = {"ran": "run", "running": "run", "runs": "run", "better": "good",
         "walked": "walk", "walking": "walk", "walks": "walk", "mice": "mouse"}
for w in ["ran", "better", "mice", "walked", "tables"]:
    print(w, "->", lemma.get(w, w))
```

```
ran -> run
better -> good
mice -> mouse
```

```
  walked -> walk
  tables -> tables
```

The irregular forms come out right, because someone listed them. *Tables* is untouched, because nobody did: the Stemming article's lookup-table caveat that every inflected form must be listed. Real lemmatizers combine lists with rules.

Regular Expressions

The Regular expression article describes one as "a sequence of characters that specifies a match pattern in text", used for find and find-and-replace and for input validation. They "originated in 1951," when Stephen Cole Kleene described regular languages with his notation of "regular events"; Ken Thompson built the notation into the QED editor and later ed, which led to grep.

CONSTRUCT	MEANING (PER THE ARTICLE)
Concatenation	elements in sequence match strings joined in order
Alternation `|`	`gray|grey` matches either
Kleene star `*`	zero or more; `+` is one or more, `?` zero or one
`.`, `[]`, `^`, `$`	any character except newline; one character from a set; start and end anchors

- **Power.** Regular expressions describe regular languages and have the same expressive power as regular grammars; they match "exactly the class of languages accepted by deterministic finite automata".
- **Warnings.** Backtracking implementations can take exponential time on patterns such as `(a|aa)*b`, which can enable a security problem called ReDoS. Features like backreferences go beyond regular languages.

A morphological use, again with my own code. This tiny analyser splits an ending off a word using patterns with a three-letter minimum for the stem:

```
import re
rules = [(r"^(\w{3,})ing$", "+ING"), (r"^(\w{3,})ed$", "+ED"), (r"^(\w{3,})s$", "+S")]

def analyse(word):
    for pat, tag in rules:
```

```
            m = re.match(pat, word)
            if m:
                return m.group(1) + tag
        return word + "+BASE"

    for w in ["walking", "walked", "walks", "walk", "bus", "sing", "need"]:
        print(w, "->", analyse(w))
```

```
walking -> walk+ING
walked -> walk+ED
walks -> walk+S
walk -> walk+BASE
bus -> bus+BASE
sing -> sing+BASE
need -> need+BASE
```

Words come out as a stem plus a label, which is the shape of a real morphological analysis. The three-letter minimum protects *bus*, *sing* and *need*, but it also means *red* and *sung* would be handled by luck: the rule knows nothing about what is a real stem.

Finite-State Ideas

The Finite-state transducer article defines an FST as a finite-state machine with two tapes, an input and an output, which maps between two sets of symbols. A finite-state automaton, with a single tape, defines a set of accepted strings; a transducer defines a *relation* between sets of strings. It adds that FSTs are often used for phonological and morphological analysis in NLP, with letters going in and morphemes coming out, and names Ronald Kaplan, Lauri Karttunen, Martin Kay and Kimmo Koskenniemi as pioneers. Weighted FSTs are used in machine translation, speech recognition, speech synthesis and optical character recognition. The article mentions Koskenniemi's 1983 two-level morphology only in its references and gives no worked example of a transducer, so I do not explain either here.

My `analyse` function above is not a finite-state transducer, but it does the same kind of job: one string in, an analysis out. Connecting it to the FST idea is my own framing.

> **WATCH OUT: RULES THAT WORK ON THE EXAMPLES YOU PICKED**
>
> Every program in this chapter works on the words I chose and fails on others I could just as easily have chosen (*running*, *this*, *studies*, *tables*). A rule set is tested on words it has not seen. Chapter 9 returns to

that.

NOTE ON SOURCES

This chapter uses Wikipedia's Stemming, Lemmatization and Finite-state transducer articles (read in full) and Regular expression (read to 100,000 of 148,323 characters; it ends partway through its Notes section). The Stemming page does not use the word "agglutinative"; the Lemmatization page has no "saw" example; the Finite-state transducer page has no worked example. NLP 1 covers stemming and lemmatization inside its text-preprocessing pipeline, from the machine-learning side. All code, outputs and the toy analysers are mine and were run with `py -X utf8`; I could not run the Porter algorithm itself, so I make no comparison between it and my stripper. Describing my analyser as doing "the same kind of job" as a transducer is my own framing.

Try It Yourself

EXERCISE 1

Run `strip_suffix` on *hoped*, *hopes* and *hoping*. What do you get, why is it wrong, and what does the Stemming article say about stems that are not real words?

View solution

EXERCISE 2

Add *tables* to the lemma dictionary and then try *chairs*. What does this show about the lookup approach, in the Stemming article's words?

View solution

EXERCISE 3

Run `strip_suffix` on *universe*, *universal*, *university*, *alumnus* and *alumni*. Compare with what the article says the Porter stemmer does with the same words, and say what you can and cannot conclude.

View solution

CHAPTER 4 QUICK REFERENCE

- Stemming: reduce to a stem by rule; stems need not be real words; over- and under-stemming happen
- Lemmatization: find the dictionary form; needs part of speech and context
- Regular expressions: patterns with concatenation, alternation and star; equivalent in power to finite automata
- Finite-state transducer: two tapes, a relation between strings; used for morphology

COMING NEXT

Computational Linguistics 5 predicts the next word by counting what usually follows what.

Computational Linguistics

Course 1 · Chapter 5 · Language Models by Counting

Phones suggest the next word, and speech recognisers choose between "recognise speech" and "wreck a nice beach". Both rely on a language model: a way of saying how likely a word is given the words before it. The oldest and simplest kind is built from nothing but counting.

> **THE IDEA IN A BOX**
>
> A language model is a machine for saying **how surprised it is** by a piece of text. A counting model learns that surprise from a corpus (Chapter 3), and it fails in an instructive way on anything it has not seen.

What a Language Model Is

Wikipedia's Language model article defines it as a computational model that predicts sequences in natural language, with uses including speech recognition, machine translation and natural language generation. Its history section runs: in the 1950s Noam Chomsky developed formal grammar-based approaches; in the 1970s Frederick Jelinek and colleagues at IBM developed statistical models using probabilistic n-grams; in the 2000s Yoshua Bengio and co-authors introduced a neural probabilistic language model; in 2010 Tomáš Mikolov and colleagues showed recurrent neural network models could substantially outperform n-gram models; and large language models, based mainly on transformers, are the most advanced form as of 2026. This chapter stays with the statistical stage; NLP 6 covers the neural sequence models.

n-grams

The N-gram article defines an n-gram as "a sequence of n adjacent symbols in a particular order"; the symbols can be letters, syllables, words, phonemes or DNA base pairs. Size 1 is a unigram, 2 a bigram, 3 a trigram. It says n-grams let bag-of-words models capture word order, and cites Shannon's 1951 examples and Google's n-gram corpus. The Language model article describes an n-gram model as one that "calculates the probability of the next word in a sequence from a fixed size window of previous words": a bigram model uses one previous word, a trigram two, an n-gram model $n - 1$.

Estimating by counting is my explanation. Neither page, in the parts I read, explains how the probabilities are computed from counts. The usual idea, which I demonstrate below, is: P(next word | previous word) = (how often the pair occurred) ÷ (how often the previous word occurred).

A bigram model of 66 words

The text is Chapter 3's invented lighthouse paragraph. The code and outputs are mine, run with `py -X utf8`.

```
import re, math
from collections import Counter

text = ("The old lighthouse stood on the cliff. The keeper climbed the stairs every nigh
        "and lit the lamp. Strong tea warmed the keeper after the climb. "
        "On stormy nights the keeper drank strong tea and watched the sea. "
        "The sea was grey. The lamp was bright. The keeper made a decision: "
        "he would stay another winter. Strong winds shook the lighthouse, "
        "but the lamp never failed.")
tokens = re.findall(r"[a-z]+", text.lower())
uni = Counter(tokens)
bi = Counter(zip(tokens, tokens[1:]))
V = len(uni)
print(len(tokens), V)

def p_bigram(w1, w2):
    return bi[(w1, w2)] / uni[w1]

after_the = {w2: bi[("the", w2)] for (w1, w2) in bi if w1 == "the"}
print(uni["the"], sorted(after_the.items(), key=lambda kv: -kv[1]))
print(round(p_bigram("the", "keeper"), 3), round(p_bigram("the", "sea"), 3), p_bigram("t
```

```
66 40
14 [('keeper', 4), ('lamp', 3), ('sea', 2), ('old', 1), ('cliff', 1), ('stairs', 1), ('c
0.286 0.143 0.0
```

The occurs 14 times. Four of those are followed by *keeper*, so the model says P(keeper | the) = 4/14 ≈ 0.286; *sea* follows twice, 0.143; and *dog*, which never follows *the* in this text, gets 0.0.

The Zero Problem

A sentence's probability, in this model, is the product of its bigram probabilities. Try a perfectly sensible sentence the model has not seen:

```
test = "the keeper watched the sea".split()
p = 1.0
for w1, w2 in zip(test, test[1:]):
    print(w1, w2, bi[(w1, w2)], round(p_bigram(w1, w2), 3))
    p *= p_bigram(w1, w2)
print(p)
```

```
the keeper 4 0.286
keeper watched 0 0.0
watched the 1 1.0
the sea 2 0.143
0.0
```

The pair *keeper watched* never occurs in the text (the keeper *drank* and *made* and *climbed*), so the whole sentence gets probability 0.0, as if it were impossible. That is the data sparsity problem. The Language model article treats it as a core problem: unseen word sequences would otherwise get zero probability, so seen probabilities are slightly lowered to make room for unseen ones; the methods it names are add-one smoothing, Good–Turing discounting and back-off models. It says skip-gram models were an attempt at "overcoming the data sparsity problem" of word n-gram models, and that neural embeddings are said to help alleviate the curse of dimensionality, which causes data sparsity as vocabulary size grows. The N-gram page, in the part I read, does not cover sparsity, smoothing or the limits of n-gram models.

Smoothing

The Additive smoothing article calls it "a technique used to smooth count data, eliminating issues caused by certain values having 0 occurrences", also called Laplace or Lidstone smoothing. For counts x_i out of N trials across d categories the smoothed estimate is $(x_i + \alpha) / (N + \alpha d)$, where $\alpha > 0$ is the pseudocount; $\alpha = 1$ is *add-one* smoothing, and common values are 0, ½ and 1. The estimate lies between the observed frequency and the uniform probability 1/d, so it is a shrinkage estimator. The article says treating an unseen event's observed zero as a probability of zero is "an oversimplification". It reports studies finding it more effective than

some other smoothing methods for tasks such as language-model-based pseudo-relevance feedback and recommender systems, and notes it is a common component of naive Bayes classifiers. Its limits, which it implies: the choice of α is partly subjective, and larger pseudocounts pull estimates toward uniform.

Applied to the bigram model, with V = 40 word types as the vocabulary size (my mapping of the article's formula: N = count of the first word, d = V):

```
def p_add1(w1, w2):
    return (bi[(w1, w2)] + 1) / (uni[w1] + V)

p = 1.0
for w1, w2 in zip(test, test[1:]):
    print(w1, w2, round(p_add1(w1, w2), 4))
    p *= p_add1(w1, w2)
print(round(p, 8))
print(round(sum(p_add1("the", w) for w in uni), 3))
```

```
the keeper 0.0926
keeper watched 0.0227
watched the 0.0488
the sea 0.0556
5.7e-06
1.0
```

The impossible pair now has probability 0.0227, so the sentence has a small but nonzero probability (5.7×10^{-6}). The last line is a check: the smoothed probabilities of every possible next word after *the* still add up to 1.0. The price is that every seen pair lost some probability: P(keeper | the) fell from 0.286 to 0.0926. With a 40-word vocabulary and only 66 words of text, add-one smoothing takes a large bite.

Perplexity

The Perplexity article defines it as "a measurement of how well a probability distribution or probability model predicts a sample": low perplexity means it predicts the sample well. It is "the exponentiation of the entropy", and a random variable with perplexity k has the same uncertainty as a fair k-sided die. For a language model, a model q is judged on a separate test sample, and better models "will tend to assign higher probabilities" to it, hence lower perplexity;

for language the article uses perplexity per word. Its word-level example is a model "as confused on test data as if it had to choose uniformly and independently among 247 possibilities for each word." It cautions that perplexity "is sometimes used as a measure of the difficulty of a prediction problem," but "it's not always an accurate representation". The Language model article's own evaluation section does not cover perplexity; it appears there only in the history section.

```
n = len(test) - 1          # four predictions
print(round(p ** (-1 / n), 1))
```

```
20.5
```

Perplexity 20.5: on this sentence the smoothed model is about as unsure as if it were choosing among roughly 20 equally likely words each time (the vocabulary has 40). I normalise by the four predictions, not five words, because a bigram model does not predict the first word; that is my choice, and other conventions exist.

WATCH OUT: PERPLEXITY NEEDS HELD-OUT TEXT, AND ITS NUMBERS MOVE

My "test" sentence is made from the same vocabulary as the training text, and the training text is 66 words. Perplexity here is only a demonstration. The article defines it on a *separate* test sample, and a toy model scored on a toy sentence cannot be compared with any other system's perplexity.

NOTE ON SOURCES

This chapter uses Wikipedia's N-gram, Additive smoothing and Perplexity articles (read in full) and Language model (read to 100,000 of 132,843 characters; the fetch reported the visible text ending partway through navigation). The N-gram page, in the part I read, does not explain counting, sparsity, smoothing or limits; the Language model page does not name the Markov assumption, give a general formula for the probability of a sequence, or discuss count-based estimation or back-off in detail. The "next word equals pair count over word count" description, the vocabulary-size mapping of the smoothing formula, the code, the 66-word text, the outputs and the choice to divide by four are mine. NLP 6 is cited from its chapter banner ("Sequence Models for Text"), not re-read.

Try It Yourself

EXERCISE 1

Using `uni` and `bi`, find P(tea | strong) and P(winds | strong). Then ask: what would the model predict after *strong* if the text had only the single sentence "Strong tea warmed the keeper"?

View solution

EXERCISE 2

Recompute the test sentence's probability and perplexity with α = 1, 0.5, 0.1 and 0.01 in (count + α) / (uni + α·V). Which α gives the lowest perplexity, and why is picking it using this same test sentence a problem?

View solution

EXERCISE 3

Starting from *the*, always choose the most frequent next word (ties go to whichever came first in the counts) for ten words. What does the output look like, and what does that say about using a bigram model to generate text?

View solution

CHAPTER 5 QUICK REFERENCE

- n-gram: n adjacent symbols; a bigram model predicts from one previous word
- Counting estimate (my explanation): pair count ÷ previous-word count
- Unseen sequences get zero, so smoothing: (count + α) / (N + αd); add-one is α = 1
- Perplexity: how well a model predicts a held-out sample; lower is better; like choosing among k equal options

COMING NEXT

Computational Linguistics 6 moves from words in a row to structure: grammars and parse trees.

Computational Linguistics

Course 1 · Chapter 6 · Grammars and Parsing

Chapter 5's model saw only the previous word. A sentence has more structure than that: *the man with the telescope* is a unit, and a program that knows it can answer questions a word-by-word model cannot. This chapter writes grammar rules in a form a machine can apply, and then meets the problem that follows every attempt: the same sentence can have more than one structure.

> **THE IDEA IN A BOX**
>
> A grammar is a small set of rewriting rules; a **parse** is the record of which rules produced a sentence. When a sentence has more than one record, the program has to *choose*.

Context-Free Grammars

Wikipedia's Context-free grammar article defines a CFG as a formal grammar whose production rules "can be applied to a nonterminal symbol regardless of its context". Formally it has four parts: nonterminals, terminals, production rules and a start symbol, which "represent[s] the whole sentence (or program)". Each rule has one nonterminal on the left and a string of terminals and/or nonterminals on the right; a nonterminal may have several rules; the language it generates is "the set of all strings of terminal symbols that can be derived" from the start symbol.

- **Where it came from.** The article says Noam Chomsky "invented" CFGs to describe natural-language sentence structure. Some linguists use "phrase structure grammar" to mean CFGs, and phrase structure grammars are also called constituency grammars, contrasted with dependency grammars. In computer science CFGs describe programming-language syntax.
- **Derivation.** "A sequence of grammar rule applications that transform the start symbol into the string"; it imposes hierarchical structure, shown as a parse tree.
- **Where it fits.** In Chomsky's classification, context-free is Type-2, with regular grammars as a subclass of linear grammars. This links to Chapter 4: regular expressions match regular languages, and context-free grammars are the next step up in power, because they can describe nesting, such as balanced parentheses, which a regular expression cannot. (The

nesting remark is the article's grammar examples, such as S → SS | (S) | (), read with Chapter 4's regular-language point; stating the contrast is my own.)

The limits for natural language

The article says context-free rules do not capture features such as agreement and reference. Chomsky's view that natural language is not context-free has held up, but his specific counterexamples were later disproved. Gazdar and Pullum argue natural language has a few non-context-free constructions, such as cross-serial dependencies in Swiss German and reduplication in Bambara, and that "the vast majority of forms in natural language are indeed context-free". The page was read to 100,000 of 107,898 characters; the unread end is mostly navigation.

Parse Trees

The Parse tree article defines one as "an ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar". There are two kinds:

	CONSTITUENCY TREE	DEPENDENCY TREE
Follows	phrase structure grammars	dependency grammars
Nodes	interior nodes labelled by non-terminal categories; leaves are terminals	all nodes terminal
Size	more nodes	fewer nodes; omit phrasal categories such as S, VP, NP, but still show constituent structure, since any complete subtree is a constituent

The article's example is *John hit the ball*: S is the root, with NP (*John*) and VP as branches; the VP contains V (*hit*) and an NP containing D (*the*) and N (*ball*). "Parse tree" is mainly the computational-linguistics term; "syntax tree" is more common in theoretical syntax. (The tree itself is an image in the article; I am relying on its text description.)

A Grammar You Can Run

The grammar, words and parser below are mine. The parser tries every way of splitting a stretch of words between a rule's two parts and remembers the results; it is meant to be short and clear, not fast. I ran it and copied the output. The bracketed lines are trees: for example `[NP i]` means the word *i* was an NP by way of the rule NP → Pro.

```
from functools import lru_cache

rules = {
    "S":  [("NP", "VP")],
    "NP": [("Pro",), ("Det", "N"), ("NP", "PP")],
    "VP": [("V", "NP"), ("VP", "PP")],
    "PP": [("P", "NP")],
}
lexicon = {
    "Pro": {"i"}, "Det": {"the"}, "N": {"man", "telescope", "ball"},
    "V": {"saw", "hit"}, "P": {"with"},
}

def parse(words, start="S"):
    words = tuple(words)

    @lru_cache(maxsize=None)
    def trees(sym, i, j):
        out = []
        if sym in lexicon:
            if j == i + 1 and words[i] in lexicon[sym]:
                out.append((sym, words[i]))
            return tuple(out)
        for rhs in rules[sym]:
            if len(rhs) == 1:
                for t in trees(rhs[0], i, j):
                    out.append((sym, t))
            else:
                for k in range(i + 1, j):
                    for left in trees(rhs[0], i, k):
                        for right in trees(rhs[1], k, j):
                            out.append((sym, left, right))
        return tuple(out)

    return trees(start, 0, len(words))

def show(t):
    if len(t) == 2 and isinstance(t[1], str):
        return t[1]
```

```
    return "[" + t[0] + " " + " ".join(show(c) for c in t[1:]) + "]"

s = "i saw the man with the telescope".split()
for t in parse(s):
    print(show(t))
print(len(parse(s)))
print(len(parse("i hit the ball".split())), len(parse("hit the i".split())))
```

```
[S [NP i] [VP saw [NP [NP the man] [PP with [NP the telescope]]]]]
[S [NP i] [VP [VP saw [NP the man]] [PP with [NP the telescope]]]]
2
1 0
```

Two parses for one sentence. In the first, *with the telescope* attaches to *the man* (the man has the telescope); in the second it attaches to the verb phrase (I used the telescope to see). The sentence *i hit the ball* has one parse and the word salad *hit the i* has none, so the grammar is also a test of whether a string belongs to its language.

Ambiguity

The Syntactic ambiguity article defines it as "the potential for a sentence to yield multiple interpretations due to its ambiguous syntax", arising from how words and clauses relate and not from individual word meanings. Its example: *John saw the man on the mountain with a telescope*, with readings in which the telescope belongs to John's seeing, to the man, or to the mountain. It distinguishes globally ambiguous sentences (*The woman held the baby in the green blanket*) from locally ambiguous ones that resolve by the end, and from garden-path sentences, where "a structurally correct sentence is difficult to interpret" because the reader commits to the wrong structure early. It also gives the headline *Violinist linked to JAL crash blossoms*. For NLP, it defines a "parse forest" as "the collection of all possible syntactic structures" for a sentence, and "syntactic disambiguation" as "the task of clarifying which meaning is actually intended". My code's list of two trees is a tiny parse forest, though the article does not use my example or my grammar.

How fast does the number of parses grow? I added more prepositional phrases with the same code:

```
base = "i saw the man"
for n in range(0, 5):
```

```
    sent = (base + " with the telescope" * n).split()
    print(n, len(parse(sent)))
```

```
0 1
1 2
2 5
3 14
4 42
```

1, 2, 5, 14, 42: each extra phrase multiplies the possibilities. Chapter 1's NLP article said a typical sentence may have thousands of potential parses; this toy grammar shows how easily they multiply. I recognise this sequence from mathematics, but neither article mentions it, so I only report the run.

Parsing in Practice

The Parsing article defines it as analyzing a string of symbols and checking that it conforms to the rules of a formal grammar. For natural language it cites "substantial ambiguity in the structure of human language" as the reason programs struggle, and says "Most modern parsers are at least partly statistical": they learn from hand-parsed corpora (Chapter 3's treebanks) using methods such as probabilistic context-free grammars, maximum entropy and neural nets; lexical statistics help but risk overfitting and need smoothing (Chapter 5). It also describes:

IDEA	WHAT THE ARTICLE SAYS
Top-down	expands rules from the start symbol, leftmost, consuming tokens left to right; LL and recursive-descent parsers
Bottom-up	starts from the input and works back to the start symbol; LR (shift-reduce) parsers
CYK	a cubic-time algorithm for context-free grammars in Chomsky normal form
Earley	a cubic-time algorithm that handles any context-free grammar
Shallow parsing	finds only the boundaries of major constituents

My parser is closest in spirit to a chart approach that tries all splits of a span, but I have not implemented CYK or Earley and do not claim it is either.

WATCH OUT: A GRAMMAR DESCRIBES WHAT YOU PUT IN IT

My grammar accepts *the telescope saw the man* (an NP can be anything, and nothing says telescopes cannot see) and has no way to parse *i saw*, since every verb needs an object. Both limits come from what I wrote, not from English. This is the article's point that context-free rules do not capture features such as agreement and reference.

NOTE ON SOURCES

This chapter uses Wikipedia's Context-free grammar (read to 100,000 of 107,898 characters), Parse tree, Syntactic ambiguity and Parsing (all read in full). The Syntactic ambiguity page does not use "I saw the man with the telescope" or the term PP-attachment, so those are mine; it gives NLP only brief coverage. The Parse tree example tree is an image, and I used the article's text description. The grammar, the parser, the sentences, all outputs and the growth sequence are mine and were run with `py -X utf8`. The link to Chapter 4 (regular versus context-free power) rests on the article's examples and my own reading. NLP 7's coverage of syntax-related tagging is not re-read here.

Try It Yourself

EXERCISE 1

Run `parse` on *the man saw i*, *i saw*, *the telescope saw the man* and *the man saw the telescope*. For each, say whether the result matches English, and name which limit of this grammar explains any mismatch.

View solution

EXERCISE 2

Add `("Name",)` to the NP rules and `"Name": {"john"}` to the lexicon, then parse *john hit the ball*. Compare your tree with the Parse tree article's description of the same sentence and list the differences in labels.

View solution

EXERCISE 3

The two parses of *i saw the man with the telescope* give two meanings. Say which meaning goes with each tree, and what kind of extra information (not in the grammar) a program would need to prefer

one.

View solution

CHAPTER 6 QUICK REFERENCE

- CFG: nonterminals, terminals, rules, start symbol; Type-2 in Chomsky's hierarchy
- Parse tree: constituency (phrase categories) or dependency (words only)
- Ambiguity: more than one tree for one string; a parse forest holds them all
- Most modern parsers are at least partly statistical and learn from treebanks

COMING NEXT

Computational Linguistics 7 labels each word with its part of speech, rule by rule and by statistics.

Computational Linguistics

Course 1 · Chapter 7 · Tagging and Sequences

Chapter 6 needed to know that *saw* was a verb and *telescope* a noun, and it got that from a hand-written lexicon. This chapter asks how a program can assign such labels to running text, when the same word can carry different labels in different sentences. It is one of the oldest tasks in the field, and it shows the choice between hand-written rules and counted statistics in miniature.

> **THE IDEA IN A BOX**
>
> Tagging gives every token one label from a fixed set. Looking the word up is easy; the work is in the words with more than one possible label, where only the *neighbours* can decide.

Parts of Speech as Labels

Wikipedia's Part of speech article defines a part of speech as a category of words with similar grammatical properties, sharing syntactic behaviour and sometimes morphological and semantic behaviour. It separates **open classes**, which "commonly accept the addition of new words" (typically nouns, verbs, adjectives, adverbs, interjections), from **closed classes** that rarely gain members (typically prepositions, determiners, conjunctions, pronouns). It says "almost all languages have the word classes noun and verb" but that other classes vary: Japanese has more adjective classes than English, and some languages lack a clear adjective class. It also says many English words can belong to more than one part of speech, with *neigh*, *break* and *telephone* as verb-or-noun examples, a shift it calls conversion or zero derivation.

The Part-of-speech tagging article defines the task as marking each word in a corpus as a particular part of speech "based on both its definition and its context". Its points about the labels themselves:

- **Tagsets are bigger than school grammar.** Schools teach about 9 English parts of speech; computational tagging typically distinguishes "50 to 150 separate parts of speech for English".
- **Different corpora, different tagsets.** The article names the Brown and LOB tagsets, says the Penn tagset is the most popular for American English and is "much smaller", and says Petrov

and colleagues proposed a universal tagset of 12 categories. It adds that smaller tagsets make automatic tagging easier.

- **Why it is hard.** Many word forms are ambiguous. Its example: *dogs* is usually a plural noun but is a verb in *The sailor dogs the hatch*.

To keep the code short, this chapter uses only seven labels: DET, NOUN, VERB, AUX, ADJ, PRON and ADP. They are my own simplification, not any published tagset.

A Tiny Corpus and a Baseline

The corpus below is mine: ten invented sentences, 48 tokens, tagged by hand. It is chosen so that *light* and *watch* are each both noun and verb, as in Chapter 3's lighthouse theme. Real taggers are trained on corpora such as the Brown Corpus (Chapter 3), thousands of times larger.

The simplest tagger gives every word its most frequent tag in the training data, and gives unseen words NOUN. I ran everything in this chapter with `py -X utf8` and copied the output.

```
from collections import Counter, defaultdict

train = """
the/DET keeper/NOUN watched/VERB the/DET light/NOUN
the/DET light/NOUN is/VERB bright/ADJ
we/PRON light/VERB the/DET lamp/NOUN
the/DET lamp/NOUN will/AUX light/VERB the/DET room/NOUN
a/DET bright/ADJ light/NOUN shone/VERB
the/DET keeper/NOUN will/AUX watch/VERB the/DET sea/NOUN
a/DET watch/NOUN hangs/VERB on/ADP the/DET wall/NOUN
we/PRON watch/VERB the/DET boats/NOUN
the/DET keeper/NOUN lit/VERB the/DET lamp/NOUN
the/DET sea/NOUN is/VERB calm/ADJ
"""
sents = [[tuple(tok.rsplit("/", 1)) for tok in line.split()]
         for line in train.strip().splitlines()]
print(len(sents), "sentences,", sum(map(len, sents)), "tokens")

wt = defaultdict(Counter)
for s in sents:
    for w, t in s:
        wt[w][t] += 1
best = {w: c.most_common(1)[0][0] for w, c in wt.items()}
for w in ("light", "watch", "bright"):
```

```
    print(w, dict(wt[w]), "->", best[w])

def baseline(words):
    return [best.get(w, "NOUN") for w in words]

test = "we light the lamp".split()
print(list(zip(test, baseline(test))))
test2 = "the keeper will watch the light".split()
print(list(zip(test2, baseline(test2))))
```

```
10 sentences, 48 tokens
light {'NOUN': 3, 'VERB': 2} -> NOUN
watch {'VERB': 2, 'NOUN': 1} -> VERB
bright {'ADJ': 2} -> ADJ
[('we', 'PRON'), ('light', 'NOUN'), ('the', 'DET'), ('lamp', 'NOUN')]
[('the', 'DET'), ('keeper', 'NOUN'), ('will', 'AUX'), ('watch', 'VERB'), ('the', 'DET'),
```

The baseline gets *watch* and *light* right in the second sentence and *light* wrong in the first (*we light the lamp*): it always says NOUN for *light*. The tagging article reports the same idea at full scale: giving each word its most common tag, and "proper noun" to unknown words, "will approach 90% accuracy" (it cites Charniak, 1997). So the baseline is strong, and the remaining errors are the ambiguous words.

Two Families of Tagger

The tagging article says algorithms fall into two groups, rule-based and stochastic. Its history gives the shape of the story:

WHEN	WHAT THE ARTICLE SAYS
Mid-1960s	The Brown Corpus, about 1,000,000 words, built by Kučera and Francis; Greene and Rubin's first program was about 70% correct, and hand correction made it nearly perfect by the late 1970s
Mid-1980s	European researchers used hidden Markov models on the Lancaster-Oslo-Bergen Corpus; the CLAWS tagger enumerated all possibilities and reached "93-95%", at high cost
1987	DeRose and Church independently developed faster Viterbi-like dynamic-programming algorithms, both reporting over 95%; the field then treated tagging as separable from other levels of NLP
Later	Machine-learning methods (support vector machines, maximum entropy, perceptron, nearest-neighbour) reported above 95% (the article marks this claim "citation needed"); one 2014 result is

WHEN	WHAT THE ARTICLE SAYS
	given as 97.36% on a standard benchmark

Rule-based: the Brill tagger

Wikipedia's Brill tagger article says Eric Brill described it in his 1993 PhD thesis, as an "error-driven transformation-based tagger". It first gives each known word its most frequent tag (and unknown words NOUN, which the article calls naive), then applies contextual rules that change tags by looking at small amounts of surrounding context, until a threshold is reached or no rule applies. Rules have the form *tag1 → tag2 IF condition*. Its example changes *while* from a preposition to a noun when the previous word is a determiner, as in *all the while*. It says rules are simple to implement as finite-state machines (compare Chapter 4) and typical taggers use "a few hundred rules", written by intuition or learned from a tagged corpus. The page does not describe rule templates in detail or list advantages, so I say nothing more about them.

Below is one rule of that style, written by me, applied after the baseline: change NOUN to VERB when the previous tag is PRON or AUX and the word has been seen as a verb.

Statistical: a hidden Markov model

The statistical alternative asks which whole tag sequence is most probable. A **bigram HMM tagger** scores a sequence by multiplying, for each word, the chance of the tag following the previous tag (Chapter 5's idea, applied to tags) by the chance of the tag producing the word. Trying every sequence would be slow; the Viterbi algorithm, named in the tagging article, finds the best one by dynamic programming. The code below is my own short version, with a small smoothing constant of 0.01 (Chapter 5) so unseen combinations do not score zero. I have not implemented CLAWS, Brill's learner or any named published tagger, and do not claim this equals them.

```
# one Brill-style rule: NOUN -> VERB after PRON or AUX
def brill(words, tags):
    tags = list(tags)
    for i in range(1, len(tags)):
        if tags[i] == "NOUN" and tags[i-1] in ("PRON", "AUX") and "VERB" in wt.get(words
            tags[i] = "VERB"
    return tags
print(list(zip(test, brill(test, baseline(test)))))

# bigram HMM with Viterbi
trans = defaultdict(Counter); emit = defaultdict(Counter)
```

```
for s in sents:
    prev = "<s>"
    for w, t in s:
        trans[prev][t] += 1; emit[t][w] += 1; prev = t
tags = sorted(emit)
def P(table, a, b, n_out):
    c = table[a]; return (c[b] + 0.01) / (sum(c.values()) + 0.01 * n_out)
V = len({w for s in sents for w, _ in s})
def viterbi(words):
    best_ = [{t: (P(trans, "<s>", t, len(tags)) * P(emit, t, words[0], V + 1), None) for
    for w in words[1:]:
        row = {}
        for t in tags:
            p, back = max((best_[-1][u][0] * P(trans, u, t, len(tags)), u) for u in tags
            row[t] = (p * P(emit, t, w, V + 1), back)
        best_.append(row)
    t = max(tags, key=lambda x: best_[-1][x][0]); out = [t]
    for row in reversed(best_[1:]):
        t = row[t][1]; out.append(t)
    return out[::-1]
for ts in (test, test2, "we watch the light".split()):
    print(list(zip(ts, viterbi(ts))))
```

```
[('we', 'PRON'), ('light', 'VERB'), ('the', 'DET'), ('lamp', 'NOUN')]
[('we', 'PRON'), ('light', 'VERB'), ('the', 'DET'), ('lamp', 'NOUN')]
[('the', 'DET'), ('keeper', 'NOUN'), ('will', 'AUX'), ('watch', 'VERB'), ('the', 'DET'),
[('we', 'PRON'), ('watch', 'VERB'), ('the', 'DET'), ('light', 'NOUN')]
```

Both fix *we light the lamp*: after *we* (a pronoun), *light* becomes a verb, whether by my one rule or by the counted tag-to-tag chances. The Viterbi tagger also keeps *watch* a verb and *light* a noun in the longer sentence.

Scoring It, and Why Not to Trust This Score

I wrote five new sentences, tagged by hand, and counted correct tags (24 tokens in all): baseline 22 of 24, baseline plus the rule 24 of 24, Viterbi 24 of 24. This does not show the methods are equally good. The test sentences reuse the training words and patterns, there are only 24 tokens, and I wrote both the sets and the rule after seeing the errors. Chapter 9 returns to how to measure honestly.

WATCH OUT: A PERFECT SCORE ON A SMALL TEST SET MEANS LITTLE

My tiny test is easy by design. The article's figures (about 90% for the baseline, over 95% for the HMM methods and later machine-learning methods) come from large corpora; mine come from five sentences I wrote. Neither my numbers nor the article's can be compared directly.

Where neighbours still fail

One more run shows that context and statistics have limits when training data is tiny:

```
w = "the lamp will watch light".split()
print("brill  ", list(zip(w, brill(w, baseline(w)))))
print("viterbi", list(zip(w, viterbi(w))))
```

```
brill   [('the', 'DET'), ('lamp', 'NOUN'), ('will', 'AUX'), ('watch', 'VERB'), ('light',
viterbi [('the', 'DET'), ('lamp', 'NOUN'), ('will', 'AUX'), ('watch', 'VERB'), ('light',
```

The Viterbi tagger calls the final *light* a preposition. I checked the counts: in the 48 training tokens, VERB is followed by a determiner 6 times, an adjective 2 times, a preposition once, and a noun never. So VERB followed directly by NOUN gets only the smoothing amount, and the tag ADP (seen only on the word *on*) can win. Which tag is "right" for this odd sentence is my judgement; the point is that the answer rests on the counts.

NOTE ON SOURCES

This chapter uses Wikipedia's Part of speech, Part-of-speech tagging and Brill tagger (each read in full; the tagging article carries a "needs more citations" banner and its unsupervised-taggers section is uncited). The 90% baseline, the 93-95% and over-95% figures and the 97.36% figure are the article's, and the "above 95%" machine-learning claim is marked "citation needed" there. The Brill page gives no rule templates or advantages. The seven-label tagset, the corpus, both taggers, the extra rule and all outputs and scores are mine, run with `py -X utf8`. I did not read the Hidden Markov model, Viterbi algorithm or Penn Treebank articles, so nothing is said about them beyond what the tagging article states. Machine-learning taggers are covered in NLP 7 (named entity recognition and part-of-speech tagging), which I have not re-read for this chapter.

Try It Yourself

EXERCISE 1

Tag *the keeper climbed the stairs* with the baseline and with `viterbi`. Which words are unseen, and how does each tagger treat them? Then write a suffix rule (a word ending in *-ed* that is unseen becomes VERB) and apply it after the baseline.

View solution

EXERCISE 2

Run `brill` and `viterbi` on *the lamp will watch light*. Explain why the two taggers disagree on the last word, and what you would add to the training data to fix it.

View solution

EXERCISE 3

The chapter's own score is 24 of 24 for two taggers. List three reasons that score is weak evidence, and say what a better test would look like.

View solution

CHAPTER 7 QUICK REFERENCE

- Tagging: one label per token, chosen from a tagset (computational tagsets typically 50-150 tags for English)
- Baseline: most frequent tag per word; unknown words guessed as noun
- Brill: start from the baseline, then apply context rules "tag1 to tag2 IF condition"
- HMM + Viterbi: find the most probable whole tag sequence by dynamic programming
- Small test sets that overlap the training data prove very little

COMING NEXT

Computational Linguistics 8 turns from labels to meaning: dictionaries of senses, and meaning drawn from the company words keep.

Computational Linguistics

Course 1 · Chapter 8 · Meaning

So far the course has dealt with forms: characters, tokens, stems, tags and trees. A program that only handles forms can say that *lamp* is a noun, but not that a lamp is more like a lantern than like a harbour. This chapter shows two ways a machine can represent meaning: by writing it down as a network of relations, and by measuring the company a word keeps.

> **THE IDEA IN A BOX**
>
> One approach *states* meaning (people build a database of senses and relations). The other *measures* it (a program counts which words appear near which). Each fails in a different place.

Meaning Written Down: WordNet

Wikipedia's WordNet article calls it "a lexical database of semantic relations between words", a combination of a dictionary and a thesaurus, created in 1985 at Princeton University's Cognitive Science Laboratory under the psychologist George Armitage Miller, and later directed by Christiane Fellbaum. Roughly synonymous words of the same part of speech are grouped into **synsets**, each with a short gloss and usage examples; a word with several senses appears in several synsets. The main relations:

RELATION	WHAT THE ARTICLE SAYS
Hypernym / hyponym	the "is a kind of" parent and its reverse (nouns and verbs)
Coordinate terms	siblings sharing a parent
Holonym / meronym	the whole and its part (nouns)
Troponym, entailment	verb relations: doing something in a particular manner; one action implying another
Antonyms	shown for adjectives, where central pairs such as hot and cold anchor "dumbbell" structures with satellite synonyms

The article lists uses in word-sense disambiguation (where it calls WordNet the most commonly used computational lexicon of English), information retrieval, text classification, summarization and machine translation, and says word similarity can be measured by distance in the graph. It also notes that other resources build on it: ImageNet is organised by WordNet's noun hierarchy.

> **WATCH OUT: THE ARTICLE GIVES TWO SETS OF COUNTS**
>
> The infobox (2024 Edition, released 1 November 2024) says 161,705 words in 120,630 synsets, while the "database contents" section says 155,327 words in 175,979 synsets (version not stated). The article does not reconcile them, so I quote neither as the size of WordNet. It also says that as of 2026 the project is no longer being developed and the online version is unavailable, with community projects such as Open English WordNet continuing the data.

A hypernym tree you can run

I have not used WordNet's data in this course (Python's `nltk` is not installed here, and I have not read its interface). Instead, the tree below is a tiny made-up hypernym network of my own, and the distance is the number of steps up to the nearest shared ancestor, added for both words. The "steps through the graph" idea is the article's; this particular measure is just the simplest one I could write. I ran it and copied the output.

```
hyper = {"lamp": "light_source", "lantern": "lamp", "candle": "light_source",
         "light_source": "artifact", "boat": "vessel", "ship": "vessel",
         "vessel": "artifact", "keeper": "person", "sailor": "person",
         "person": "entity", "artifact": "entity"}
def path_up(w):
    out = [w]
    while hyper.get(w):
        w = hyper[w]; out.append(w)
    return out
def dist(a, b):
    pa, pb = path_up(a), path_up(b)
    common = next(x for x in pa if x in pb)
    return pa.index(common) + pb.index(common)
for a, b in [("lantern", "lamp"), ("lantern", "candle"), ("boat", "ship"),
             ("lantern", "boat"), ("lantern", "keeper")]:
    print(a, b, dist(a, b))
print(path_up("lantern"))
```

```
lantern lamp 1
lantern candle 3
```

```
boat ship 2
lantern boat 5
lantern keeper 6
['lantern', 'lamp', 'light_source', 'artifact', 'entity']
```

Closer in the tree means more alike: a lantern is one step from a lamp, three from a candle, six from a keeper. Everything this tells us was put there by whoever drew the tree, which is both the strength (it is deliberate and inspectable) and the cost (someone must build it, word by word).

Choosing a Sense: Word-Sense Disambiguation

Wikipedia's Word-sense disambiguation article defines it as "the process of identifying which sense of a word is meant in a sentence or other segment of context", usually subconscious in people. Its examples include *pen* (writing instrument or enclosure) and *bank*, which a translator might render as French *banque* or *rive*. It names Lesk (1986) as "the seminal dictionary-based method": find "the pair of dictionary senses with the greatest word overlap" in their definitions. It groups approaches into knowledge-based (dictionaries and thesauri, "without using any corpus evidence"), supervised (classifiers trained on sense-tagged corpora, held back by a "knowledge acquisition bottleneck"), semi-supervised and unsupervised (clustering occurrences, "the greatest challenge"). Its figures:

- Coarse-grained, homograph-level English WSD "routinely above 90%" (as of 2009); fine-grained top results "from 59.1% to 69.0%" in SemEval-2007 and Senseval-2.
- In Senseval-2, with fine-grained senses, human annotators "agreed in only 85% of word occurrences", so human performance is treated as an upper bound.
- Always choosing the most frequent sense scored 51.4% (SemEval-2007) and 57% (Senseval-2), showing how much comes from frequency alone.
- History: Warren Weaver raised it as a computational problem in 1949; the 1990s statistical revolution made it a supervised-learning problem.

I read the article to 100,000 of 113,521 characters. Below is a very simplified overlap method in the spirit of Lesk's idea, with two glosses I wrote for *light*. It is my sketch, not Lesk's algorithm as published, which I have not read.

```
senses = {
    "light": {
        "lamp-light": "brightness from a lamp or the lighthouse that guides boats at nigh
```

```
        "not-heavy": "of small weight easy to carry not heavy",
    }
}
stop = {"the", "a", "of", "or", "to", "at", "in", "is", "and", "from", "that"}
def lesk(word, context):
    ctx = {w for w in context.split() if w not in stop}
    scores = {s: len(ctx & {w for w in g.split() if w not in stop})
              for s, g in senses[word].items()}
    return scores, max(scores, key=scores.get)
for c in ("the light guides the boats at night", "the bag is light and easy to carry", "
    print(c, "->", lesk("light", c))
```

```
the light guides the boats at night -> ({'lamp-light': 3, 'not-heavy': 0}, 'lamp-light')
the bag is light and easy to carry -> ({'lamp-light': 0, 'not-heavy': 2}, 'not-heavy')
```

The first two work because their context words overlap a gloss. In the third there is no overlap at all, so `max` simply returns the first sense: the program has no evidence but still answers. A real system must be able to say "I can't tell", or fall back on a baseline such as the most frequent sense.

Meaning Measured: Distributional Semantics

The Distributional semantics article describes it as a research area that measures and classifies semantic similarity between linguistic items using their distributional properties in large corpora, built on the distributional hypothesis: "linguistic items with similar distributions have similar meanings." It attributes the phrase "a word is characterized by the company it keeps" to Firth, who popularised it in the 1950s, and cites Harris (1954) for the hypothesis. It gives no longer Firth quotation and no Harris wording, so I quote nothing more. Its other points:

- Co-occurrence information is collected in high-dimensional vectors, and similarity is defined as vector similarity; when the context is other words the models are called "word space" or "vector space" models.
- It lists cosine similarity and Minkowski distance as similarity measures, and dimension reduction (random indexing, singular value decomposition) as one parameter; it explains neither method.

- Models differ in context type, window size, frequency weighting, dimension reduction and similarity measure; latent semantic analysis, Hyperspace Analogue to Language and topic models are named.
- It says distributional semantics serves as an input to neurally inspired deep-learning models, and lists word embedding and Word2vec only in "See also". NLP 5 (Word Embeddings and Word2Vec) covers that route, and I have not re-read it.
- Limits it admits: it calls it an open question whether similar meaning really tracks similar distribution; it mentions data sparsity; and it says word-level meaning is only part of a sentence's meaning, with *Tigers love rabbits* as its example.

Wikipedia's Cosine similarity article gives the measure: "the dot product of the vectors divided by the product of their lengths", from −1 to +1, and 0 to 1 when no component can be negative (as with counts). It says that for documents it works as a way of "normalizing document length". Below is my own tiny corpus (12 invented sentences) and a window of two words either side, ignoring *the*.

```
import math
from collections import Counter, defaultdict

text = """
the keeper lit the lamp
the keeper lit the lantern
the sailor watched the lamp
the sailor watched the lantern
the keeper climbed the stairs
the sailor climbed the mast
the sailor rowed the boat
the sailor rowed the ship
the keeper watched the boat
the keeper watched the ship
the boat reached the harbour
the ship reached the harbour
"""
sents = [l.split() for l in text.strip().splitlines()]
vec = defaultdict(Counter)
for s in sents:
    for i, w in enumerate(s):
        for j in range(max(0, i-2), min(len(s), i+3)):
            if j != i and s[j] != "the":
                vec[w][s[j]] += 1
def cos(a, b):
```

```
    dot = sum(a[k] * b[k] for k in a)
    return dot / (math.sqrt(sum(v*v for v in a.values())) * math.sqrt(sum(v*v for v in b
print("lamp", dict(vec["lamp"]))
print("lantern", dict(vec["lantern"]))
for a, b in [("lamp", "lantern"), ("boat", "ship"), ("lamp", "boat"),
             ("keeper", "sailor"), ("lamp", "keeper")]:
    print(a, b, round(cos(vec[a], vec[b]), 3))
```

```
lamp {'lit': 1, 'watched': 1}
lantern {'lit': 1, 'watched': 1}
lamp lantern 1.0
boat ship 1.0
lamp boat 0.408
keeper sailor 0.556
lamp keeper 0.943
```

Lamp and *lantern* score 1.0 and *boat* and *ship* score 1.0, which is the pattern the hypothesis predicts: I wrote them to appear in identical contexts, so this is partly built in. The surprise is the last line: *lamp* and *keeper* score 0.943, higher than *keeper* and *sailor* (0.556), though a keeper is a person and a lamp is not. The reason is in the counts: both are only ever seen next to *lit* and *watched* in my twelve sentences. The measure is faithfully reporting a corpus that is too small and too repetitive, which echoes the article's data sparsity point. I did not test how the score changes with a larger corpus.

	WRITTEN DOWN (WORDNET-STYLE)	MEASURED (DISTRIBUTIONAL)
Source	people build the network	counts from a corpus
Strength	explicit relations such as is-a; inspectable	no hand-building; reflects usage in the corpus
here	a gap means no answer	similarity is not a named relation

The table's two columns are my summary of the two sections, not a comparison made by the articles.

NOTE ON SOURCES

This chapter uses Wikipedia's WordNet and Cosine similarity (read in full), Word-sense disambiguation (read to 100,000 of 113,521 characters; the tail was not read) and Distributional semantics (read in full;

it has no criticisms section and no formula for cosine). The Firth phrase is the only wording the page gives; I did not verify any longer quotation, and I did not read Firth or Harris. The WordNet counts conflict inside the article and are not used. The hypernym tree, glosses, corpus, window size, both algorithms and all outputs are mine and were run with `py -X utf8`; the Lesk-style code is not Lesk's published algorithm. NLP 5 was not re-read here.

Try It Yourself

EXERCISE 1

Add `"torch": "light_source"` to the hypernym tree. Predict, then compute, `dist` from *torch* to *lamp*, *lantern* and *candle*. Explain why *torch* is not closer to *lantern* than to *lamp*.

View solution

EXERCISE 2

Print the vectors for *keeper* and *lamp* and explain the 0.943 score. Then add the sentences *the keeper slept* and *the lamp glowed* twice each, rebuild the vectors and rerun the cosines. What changes, including for *lamp* and *lantern*?

View solution

EXERCISE 3

Change `lesk` so that when the two top scores are equal it returns `"unsure"`. Test it on *she took the light* and *the light guides the boats at night*, then name one other fallback the WSD article's baseline suggests.

View solution

CHAPTER 8 QUICK REFERENCE

- WordNet: synsets (near-synonyms) joined by relations such as hypernym/hyponym and meronym/holonym

- Word-sense disambiguation: pick the sense that fits the context; Lesk compares context with dictionary definitions
- Distributional hypothesis: similar distributions suggest similar meanings; Firth: "a word is characterized by the company it keeps"
- Cosine similarity: dot product over lengths; 0 to 1 for counts
- Small corpora give misleading similarities

COMING NEXT

Computational Linguistics 9 asks how to tell whether any of these systems is actually good: precision, recall, baselines and agreement between human judges.

Computational Linguistics

Course 1 · Chapter 9 · Measuring Success

Chapters 7 and 8 quoted accuracy figures such as "over 95%" for taggers and "51.4%" for always choosing the most frequent word sense. A number like that means little until you know what was counted, what the easy alternative scores, and whether human judges agree on the right answer. This chapter covers those three questions.

> **THE IDEA IN A BOX**
>
> A score answers a question only when you can say *what it counts*, *what a lazy system would get* and *how far people agree on the answer key*.

Counting Right and Wrong

Wikipedia's Precision and recall article starts from four counts for a yes/no decision: true positives (TP, a positive prediction that matches an actual positive), false positives (FP, a false alarm), false negatives (FN, a miss) and true negatives (TN, a correct rejection). From these:

MEASURE	FORMULA	IN WORDS (THE ARTICLE'S WORDING)
Precision	TP / (TP + FP)	"the fraction of relevant instances among the retrieved instances"
Recall	TP / (TP + FN)	"the fraction of relevant instances that were retrieved"
F1	2 · P · R / (P + R)	"the harmonic mean of precision and recall"
Accuracy	(TP + TN) / all	the share of all decisions that were right

The article says precision and recall often pull in opposite directions: perfect recall is trivial (retrieve everything) and perfect precision is easy (select only a few very likely items). It notes accuracy "can be a limited/misleading metric" when classes are unbalanced: with 95 negatives and 5 positives, predicting everything negative gives 0.95. Its worked example is a photo with 10 cats and 12 dogs, where a program picks out 8 dogs of which 5 are right: TP 5, FP 3, FN 7, TN 7, so precision 0.625 and recall about 0.417. Wikipedia's F-score article adds the general F-beta, where recall counts β^2 times as much as precision ($\beta = 2$ favours recall, $\beta = 1/2$ favours precision), and the form F1 = 2TP / (2TP + FP + FN). It also lists criticisms: F-measures "do not

take true negatives into account", F1 gives precision and recall equal weight, and the score depends on the ratio of positive to negative cases. It says the F-score is common in NLP, for example for named entity recognition and word segmentation.

An evaluation you can run

The 20 sentences, their labels and the keyword detector below are mine. The task: decide whether a sentence is about a lighthouse (5 of the 20 are). I wrote the labels by hand, which is also a weakness; see agreement, below. Before trusting my functions, I fed them the article's dog counts (the last line of the output) and got its 0.625 and 0.417.

```
data = [
 ("the keeper lit the lamp", 1), ("the light turned all night", 1),
 ("the tower stood on the rocks", 1), ("the keeper climbed the stairs", 1),
 ("ships saw the beam from far out", 1), ("she carried a light bag", 0),
 ("the market sold fresh bread", 0), ("it was a light rain", 0),
 ("we walked to the station", 0), ("the train was late again", 0),
 ("he read a book on the sofa", 0), ("the cat slept by the stove", 0),
 ("the bakery opens at six", 0), ("the lamp on my desk broke", 0),
 ("rain fell on the town", 0), ("they ate soup for lunch", 0),
 ("the bus stopped near the park", 0), ("a child drew a red kite", 0),
 ("the teacher marked the essays", 0), ("the river froze in january", 0),
]
def detect(s):
    return int(any(w in s.split() for w in ("lamp", "light", "keeper", "lighthouse")))
def counts(gold, pred):
    tp = sum(g == 1 and p == 1 for g, p in zip(gold, pred))
    fp = sum(g == 0 and p == 1 for g, p in zip(gold, pred))
    fn = sum(g == 1 and p == 0 for g, p in zip(gold, pred))
    tn = sum(g == 0 and p == 0 for g, p in zip(gold, pred))
    return tp, fp, fn, tn
def scores(tp, fp, fn, tn):
    p = tp / (tp + fp) if tp + fp else 0.0
    r = tp / (tp + fn) if tp + fn else 0.0
    f1 = 2 * p * r / (p + r) if p + r else 0.0
    return p, r, f1, (tp + tn) / (tp + fp + fn + tn)
gold = [g for _, g in data]
pred = [detect(s) for s, _ in data]
print("tp fp fn tn:", counts(gold, pred))
print("precision, recall, F1, accuracy:", [round(x, 3) for x in scores(*counts(gold, pre
for (s, g), p in zip(data, pred):
    if g != p:
        print("wrong:", s, "gold", g, "pred", p)
```

```
none = [0] * len(gold)
print("always-no:", counts(gold, none), [round(x, 3) for x in scores(*counts(gold, none)
print("dog check:", [round(x, 3) for x in scores(5, 3, 7, 7)])
```

```
tp fp fn tn: (3, 3, 2, 12)
precision, recall, F1, accuracy: [0.5, 0.6, 0.545, 0.75]
wrong: the tower stood on the rocks gold 1 pred 0
wrong: ships saw the beam from far out gold 1 pred 0
wrong: she carried a light bag gold 0 pred 1
wrong: it was a light rain gold 0 pred 1
wrong: the lamp on my desk broke gold 0 pred 1
always-no: (0, 0, 5, 15) [0.0, 0.0, 0.0, 0.75]
dog check: [0.625, 0.417, 0.5, 0.545]
```

The detector's accuracy is 0.75, which sounds respectable. But the lazy system that says "no" to every sentence *also* scores 0.75, because 15 of the 20 sentences are negatives, so the detector has gained nothing in accuracy. Only precision (0.5) and recall (0.6) reveal what it is doing: three of its six alarms are wrong (*light bag*, *light rain*, *lamp on my desk*, which are the word sense problem of Chapter 8) and it misses two lighthouse sentences that contain none of its words.

Baselines

A **baseline** is the simple system any new method must beat. The always-no rule above is one. Earlier chapters met others on real data, all from the articles I read: giving each word its most common tag and unknown words the proper-noun tag "will approach 90% accuracy" (Part-of-speech tagging, citing Charniak, 1997), so a tagger at 93-95% has to be read against that; and always choosing the most frequent sense scored 51.4% (SemEval-2007) and 57% (Senseval-2) in Word-sense disambiguation. This is why Chapter 7 treated its own 24/24 with suspicion. The term "baseline" as a general concept comes from those pages' usage and my own explanation; I did not read a separate article on it.

WATCH OUT: DO NOT TUNE ON YOUR TEST SET

A detector built after looking at the errors will look better than it is. In Exercise 1 I change the keyword list after seeing the five mistakes above, and the score rises; that rise says little about new text, because the test sentences shaped the fix. Keep a test set apart from the data you tune on.

Do the Judges Agree?

The answer key (`gold`) was written by one person. Wikipedia's Inter-rater reliability article defines the field as "the degree of agreement among independent observers who rate, code, or assess the same phenomenon". Its simplest measure, joint probability of agreement, is the percentage of cases where raters agree; the weakness is that it ignores chance, so with few categories two raters can agree "by pure chance" often. The kappa family corrects for that: Cohen's kappa for two raters, Fleiss' kappa for any fixed number. The article gives no numeric cut-offs.

Wikipedia's Cohen's kappa article gives $\kappa = (p_o - p_e) / (1 - p_e)$, where p_o is the observed proportion of agreement and p_e the agreement expected by chance from each rater's own label proportions. It ranges from −1 to 1, with 0 meaning agreement no better than chance. Its worked example (50 grant applicants, Yes/No; 20 both-Yes, 5, 10, 15 both-No) gives $p_o = 0.7$, $p_e = 0.5$ and $\kappa = 0.4$; my function reproduces that (first output line). The article also lists the Landis and Koch bands (for example 0.61-0.80 "substantial") and Fleiss's, but calls such guidelines arbitrary, based on opinion rather than evidence. Its limitations include that kappa can mislead with rare categories.

```
def kappa(a, b):
    n = len(a)
    po = sum(x == y for x, y in zip(a, b)) / n
    pe = sum((a.count(c) / n) * (b.count(c) / n) for c in set(a) | set(b))
    return po, pe, (po - pe) / (1 - pe)

# the Cohen's kappa article's 50-applicant table
A = ["Y"] * 20 + ["Y"] * 5 + ["N"] * 10 + ["N"] * 15
B = ["Y"] * 20 + ["N"] * 5 + ["Y"] * 10 + ["N"] * 15
print("article check:", [round(x, 3) for x in kappa(A, B)])

# my 20 sentences: annotator B disagrees with my labels on two
A = gold
B = list(A); B[2] = 0; B[13] = 1
print("agree", sum(x == y for x, y in zip(A, B)), "of", len(A))
print("po, pe, kappa:", [round(x, 3) for x in kappa(A, B)])

# annotator C says "yes" to just two sentences
C = [0] * 20; C[0] = 1; C[5] = 1
print("annotator C:", sum(x == y for x, y in zip(A, C)), "of 20;", [round(x, 3) for x in
```

```
article check: [0.7, 0.5, 0.4]
agree 18 of 20
po, pe, kappa: [0.9, 0.625, 0.733]
annotator C: 15 of 20; [0.75, 0.7, 0.167]
```

B agrees on 18 of 20 (90%) and gets κ ≈ 0.73. Annotator C agrees on 15 of 20 (75%) but κ is only about 0.17: because 75% of the labels are "no", chance alone would produce 70% agreement, so only a little of C's agreement is real. The 75% agreement figure and the 0.75 accuracy of always-no are the same arithmetic seen from two sides. The invented annotators B and C are only to show the maths; no real annotators were used.

Why this matters for the field: the Word-sense disambiguation article says human annotators in Senseval-2 "agreed in only 85% of word occurrences" with fine-grained senses and treats human performance as an upper bound, so a system cannot sensibly be asked to beat the people who made the key.

NOTE ON SOURCES

This chapter uses Wikipedia's Precision and recall, F-score and Inter-rater reliability (each read in full) and Cohen's kappa (read to 100,000 of 105,852 characters; the unread tail was navigation per the fetch). The fetch tool flagged one sentence in Precision and recall (about an all-negative classifier and "100% recall") as inconsistent; I did not use it. The baseline figures are from the Part-of-speech tagging and Word-sense disambiguation pages as read in Chapters 7 and 8. The 20 sentences, labels, detector, annotators B and C, and every output are mine and were run with `py -X utf8`. I did not read an article on baselines, nor on micro/macro averaging beyond the F-score page's brief mention.

Try It Yourself

EXERCISE 1

Replace the keyword list with *keeper*, *tower*, *beam* and *lighthouse*, and recompute the counts and scores. Why is the improvement not good evidence that the new detector is better?

View solution

EXERCISE 2

Using the article's F-beta formula, compute F1, F2 and F0.5 for the original detector. For a tool that flags sentences for a human to read, which would you report, and why?

View solution

EXERCISE 3

Compute raw agreement and kappa between the answer key and an annotator who labels every sentence "no". Explain the result.

View solution

CHAPTER 9 QUICK REFERENCE

- Precision = TP / (TP + FP); recall = TP / (TP + FN); F1 = harmonic mean of the two
- Accuracy can mislead when one class is common: compare with the always-majority baseline
- Baseline: the simple method a new system must beat (most frequent tag, most frequent sense)
- Kappa = $(p_o - p_e) / (1 - p_e)$: agreement corrected for chance
- Do not tune on the test set

COMING NEXT

Computational Linguistics 10 looks at what the field cannot yet do, who it affects, and where to go from here, including the NLP and LLMs courses.

Computational Linguistics

Course 1 · Chapter 10 · Limits and Society

The methods in this course work on text that someone has already typed, spelt in a standard way, in a language with plenty of data. This last chapter looks at the places where those conditions fail: spoken language, translation between languages, and speakers whose way of talking is under-represented. It ends with a recap of the course and pointers to where to go next.

> **THE IDEA IN A BOX**
>
> A language technology is only as good as the language it was built and scored on. If the data and the answer key come from one kind of speaker, the results describe that kind of speaker.

Speech: From Sound to Text

Wikipedia's Speech recognition article calls it a sub-field of computational linguistics that develops methods to "translate spoken language into text or other interpretable forms". Its history, briefly: Bell Labs' Audrey recognised single-speaker digits in 1952; hidden Markov models and n-gram language models (Chapters 5 and 7) took over in the 1980s; deep neural networks and related methods drove large error reductions in the 2000s and 2010s; and in 2017 Microsoft reported human parity on the Switchboard task. The classic design has an **acoustic model** (audio turned into spectral features about every 10 ms) and a **language model** that constrains plausible word sequences; the article says end-to-end neural models can learn these parts together.

It defines the standard score, **word error rate**, as (s + d + i) / n: substitutions, deletions and insertions over the number of words in the reference. It gives vocabulary-size figures (recognising the ten digits "essentially perfectly", while vocabularies of 200, 5,000 and 100,000 words may have error rates of 3%, 7% and 45%) and lists accent and dialect among the factors that affect accuracy. I read the article to 100,000 of 226,473 characters, so it may say more on accents and demographics later; in the part I read it gives no figures for gender or age.

Below is my own implementation of word error rate (edit distance over words), checked on a few cases, followed by a deliberately simple "recogniser" that only knows a short list of words. The

word list and the Scots-speaking example are mine; this shows the mechanism, not how any real system performs.

```
def wer(ref, hyp):
    r, h = ref.split(), hyp.split()
    d = [[0] * (len(h) + 1) for _ in range(len(r) + 1)]
    for i in range(len(r) + 1): d[i][0] = i
    for j in range(len(h) + 1): d[0][j] = j
    for i in range(1, len(r) + 1):
        for j in range(1, len(h) + 1):
            cost = 0 if r[i-1] == h[j-1] else 1
            d[i][j] = min(d[i-1][j] + 1, d[i][j-1] + 1, d[i-1][j-1] + cost)
    return d[len(r)][len(h)] / len(r)
print(wer("the keeper lit the lamp", "the keeper lit the lamp"))
print(wer("the keeper lit the lamp", "the keeper lit a lamp"))
print(wer("the keeper lit the lamp", "the keeper lamp"))
print(wer("the keeper lit the lamp", "the keeper lit the old lamp"))

known = set("i do not know the lamp is out it was a good night we went home".split())
def toy_recognise(said):
    return " ".join(w if w in known else "<unk>" for w in said.split())
pairs = [
    ("i do not know", "i do not know"),
    ("i do not know", "i dinnae ken"),
    ("it was a good night", "it was a guid nicht"),
]
for ref, said in pairs:
    out = toy_recognise(said)
    print(repr(said), "->", repr(out), "WER", round(wer(ref, out), 2))
```

```
0.0
0.2
0.4
0.2
'i do not know' -> 'i do not know' WER 0.0
'i dinnae ken' -> 'i <unk> <unk>' WER 0.75
'it was a guid nicht' -> 'it was a <unk> <unk>' WER 0.4
```

The first four lines match the formula by hand: one substitution in five words is 0.2, two deletions is 0.4, one insertion is 0.2. In the second block, the same meaning, said by a speaker who uses Scots words (*dinnae* for *do not*, *ken* for *know*, *guid* for *good*, *nicht* for *night*; the glosses are from my own knowledge and not from the articles), gets a much worse score. The toy

recogniser has not failed through any defect in the speaker; its word list was drawn from one variety. Exercise 2 shows that a second problem hides in the answer key.

Translation Between Languages

Wikipedia's Machine translation article defines it as "the use of computational techniques to translate text or speech from one language to another" and says differences in language, grammar, tone and culture make it "inadequate to replace real translators fully". The milestones it gives: Warren Weaver's 1949 memorandum, "perhaps the single most influential publication in the earliest days of machine translation"; the 1954 Georgetown-IBM demonstration; the 1966 ALPAC report, which found a decade of research had not met expectations and led to greatly reduced funding; a shift to statistical models from the late 1980s; and, in the 2020s, neural machine translation and large language models superseding earlier methods. It describes three approaches:

APPROACH	WHAT THE ARTICLE SAYS
Rule-based	everything must be made explicit, including a lexical selection rule for every ambiguity (Chapter 8's word senses)
Statistical	trained on bilingual corpora such as the Canadian Hansard and EUROPARL; depends on parallel texts; struggles with morphology-rich languages, especially translating into them (Chapter 4)
Neural	deep-learning based; the article says the "so-called human parity achieved is not real", because claims rest on limited domains, language pairs and benchmarks

Among its known problems it lists word-sense and syntactic disambiguation (raised by Bar-Hillel in the 1950s), and says machine translation handles slang, dialect and casual speech less accurately than standard language. It names BLEU, NIST, METEOR and LEPOR as automatic metrics and says human judges are the most reliable but slow; Chapter 9's agreement question applies to those judges. It also says human post-editing is usually still needed, and that medical and legal use carries risks, with some courts prohibiting it in formal proceedings. I read this article to 100,000 of 145,834 characters, and the part I read does not discuss gender bias in translation.

Who Gets Left Out

Wikipedia's Algorithmic bias article describes it as "systematic and repeatable errors that create unfair outcomes", and separates pre-existing bias (from underlying social and institutional ideologies), technical bias (from limits of a program or its design) and emergent bias (from use in new contexts). Its language-related claims, which the page carries a maintenance notice warning may include text produced by a large language model, so they should be checked against the sources it cites (I have not done that):

- English-trained models often "present the Anglo-American views as truth", and models may be biased against people based on their dialect.
- A 2024 study is described as finding that models show more negative stereotypes about speakers of African American English than recorded human biases; some hate-speech detection tools were reported 2.2 times more likely to flag content written in that dialect.
- Speech recognition accuracy can vary by accent, possibly due to "a lack of training data for speakers of that accent".
- Machine translation shows "a strong tendency towards male defaults", especially in STEM fields.
- Large language models may link nurses or secretaries with women and engineers or CEOs with men.

Each of these is the article's report. Whether it is true in a particular case needs the measures from Chapter 9, applied separately to each group of speakers: an overall score can be high while one group's is poor. Exercise 3 does this on a tiny invented example.

> **WATCH OUT: AN AVERAGE HIDES A GAP**
>
> A system with a good overall score can still do badly for a minority of its users. To find it you must break the score down by speaker group, and you need an answer key that does not itself treat one variety as the only correct one.

Links to the sociolinguistics course

Sociolinguistics: Language, Dialect and Society 2 (Dialect and Accent) treats the line between language and dialect as partly social. Sociolinguistics: Language, Dialect and Society 8 (Standards and Attitudes) covers how standard languages are made and how attitudes to accents arise. In computational terms, a "standard" is whatever the training data and the answer key contain. I am drawing that link myself; neither Wikipedia article makes it.

Course Complete: What You Can Now Do

CHAPTER	WHAT YOU MET
1 Language as Data	text as bytes and code points; normalisation; tasks of the field
2 Words and Tokens	tokenisers; types and tokens; Zipf's law
3 Corpora	corpora; concordances; collocations
4 Morphology by Machine	stemming, lemmatisation, finite-state ideas
5 Language Models by Counting	n-grams, smoothing, perplexity
6 Grammars and Parsing	context-free grammars, parse trees, ambiguity
7 Tagging and Sequences	part-of-speech tagging; rule-based and statistical taggers
8 Meaning	WordNet, sense disambiguation, co-occurrence vectors
9 Measuring Success	precision, recall, F-score, baselines, kappa
10 Limits and Society	speech, translation, bias

Across the course, the same pattern recurred: a simple method works on neat data, a limit appears, and a measurement tells you how bad it is. Everything was written in short standard-library Python on small invented data, so none of the numbers say how well any real system performs.

Where to go next: NLP picks up the same tasks with libraries and machine learning (its Chapter 1 covers the text-preprocessing pipeline, Chapter 5 word embeddings, Chapter 7 named-entity recognition and part-of-speech tagging; I have only read the chapter titles). LLMs covers how large language models are built; for example, LLMs 2 is on subword tokenisation (compare Chapter 2) and LLMs 10 on context windows and limitations. Those are the headline titles from the courses' banners, not a detailed check of their contents.

NOTE ON SOURCES

This chapter uses Wikipedia's Speech recognition (read to 100,000 of 226,473 characters), Machine translation (to 100,000 of 145,834) and Algorithmic bias (to 100,000 of 235,344, with a notice that parts may be LLM-written). I looked for a Low-resource language article; that URL returned 404, so nothing is said on it. I did not verify the bias claims against their cited studies, and I read no article on Scots or on any specific recogniser. The WER code, the toy recogniser, its word list, the Scots examples and

glosses, the numbers and the recap are mine; every output was run with `py -X utf8`. The sociolinguistics link is mine. NLP and LLMs are cited from their banners only.

Try It Yourself

EXERCISE 1

Work out by hand the word error rate of the hypotheses *the lamp was out*, *the lamp out* and *the old lamp is out* against the reference *the lamp is out*, naming each substitution, deletion or insertion. Then check with `wer`.

View solution

EXERCISE 2

Add *dinnae*, *ken*, *guid* and *nicht* to `known` and rerun the three pairs. The unknown words disappear; does the WER? Explain what the remaining score is measuring, and what you would change in the answer key.

View solution

EXERCISE 3

With the original word list, score the four items (two from group A speaking standard forms, two from group B using the Scots forms) and print the average WER per group. What does the overall average hide?

View solution

CHAPTER 10 QUICK REFERENCE

- Word error rate = (substitutions + deletions + insertions) / words in the reference
- Speech recognition: acoustic model plus language model, or end-to-end neural models
- Machine translation: rule-based, statistical, neural; non-standard speech is handled less accurately; "human parity" claims are contested

- Report scores per speaker group, not only overall
- Training data and answer keys carry a standard with them

COURSE COMPLETE

You have finished *Computational Linguistics*. For the machine-learning versions of these tasks, go on to *NLP* and *LLMs*.