

COURSE

# Words by the Numbers

10 Chapters

Philip Osztromok

Generated with Claude

WORDS BY THE NUMBERS

# Contents


| | | |
|---|---|---|
| **Chapter 1** | Counting Words | 2 |
| **Chapter 2** | Zipf's Law | 10 |
| **Chapter 3** | Letters and Secrets | 19 |
| **Chapter 4** | Neighbouring Words | 30 |
| **Chapter 5** | Markov Text | 38 |
| **Chapter 6** | Style and Authorship | 47 |
| **Chapter 7** | Readability | 58 |
| **Chapter 8** | Important Words | 69 |
| **Chapter 9** | Squeezing English | 80 |
| **Chapter 10** | Text Detective | 93 |

# Words by the Numbers 🏆

Chapter 1 · Counting Words

Before computers could read, people counted. Medieval friars indexed the Bible by hand, and later scholars tallied the words of poets by the thousand. Counting sounds dull, but it is the first move in almost everything this course does: spotting a language, naming an author, scoring how hard a page is to read, or finding the words that matter in a document. This first chapter treats a piece of writing as data, cuts it into words, counts them, and builds the small counter that later chapters keep reusing. You need arithmetic only; the maths stays light, and every idea is something you can run.

**WHAT WAS RUN, AND WHAT TO TRUST**

Every Python snippet in this chapter and in the three exercise solutions was executed, and the output in the comments is what Python printed. Background facts come from a web search that returned summaries of encyclopedia and reference pages (the type–token distinction, the history of concordances); such fetched pages are often truncated or model-summarised, so treat the wording as a paraphrase, not a quotation. The sample sentences are either the public-domain opening of an 1813 novel (check the exact wording against an edition) or short texts invented for the exercises. Every count below describes those few lines only, so it is *illustrative*, not a finding about English. Statements marked (check) were not confirmed against a source.

## Text as Data

A computer sees a passage as a long string of characters. To count anything meaningful we have to decide what the units are. Word counting needs two distinctions that look pedantic and turn out to matter all the way through the course. A word **token** is one occurrence of a word in the text; a word **type** is a distinct word, however many times it occurs. The terms come from the philosopher C. S. Peirce, who used "the" as an illustration: one word in the language, many physical copies of it on a page (this is the reference accounts' summary, not a quotation). In the sentence "the cat sat on the mat" there are six tokens and five types, because "the" is one type with two tokens.

| TERM | COUNTS | "THE CAT SAT ON THE MAT" |
|---|---|---|
| Token | every occurrence | 6 |

| TERM | COUNTS | "THE CAT SAT ON THE MAT" |
|---|---|---|
| Type | each distinct word once | 5 |
| Frequency | how many tokens a type has | the = 2, every other type = 1 |
| Vocabulary | the set of types | the, cat, sat, on, mat |

Our sample for the whole chapter is the opening of Jane Austen's *Pride and Prejudice* (1813, in the public domain): two sentences, short enough to check by eye. The first attempt at cutting it into words is the one most beginners try, splitting on spaces.

```
PASSAGE = (
    "It is a truth universally acknowledged, that a single man in possession "
    "of a good fortune, must be in want of a wife. However little known the "
    "feelings or views of such a man may be on his first entering a "
    "neighbourhood, this truth is so well fixed in the minds of the "
    "surrounding families, that he is considered the rightful property of "
    "some one or other of their daughters."
)

# First attempt: split on whitespace and count.
words = PASSAGE.split()
print("pieces:", len(words))
print(words[:8])
print([w for w in words if not w.isalpha()])
# pieces: 70
# ['It', 'is', 'a', 'truth', 'universally', 'acknowledged,', 'that', 'a']
# ['acknowledged,', 'fortune,', 'wife.', 'neighbourhood,', 'families,', 'daughters.']
```

Seventy pieces, but six of them carry punctuation: `acknowledged,` is not the same string as `acknowledged`, so a program would count it as a different word. This is the heart of **tokenising**: deciding where one word ends and the next begins. There is no single correct answer, only a choice that suits the question. A second attempt lower-cases everything and keeps runs of letters and apostrophes.

```
import re
from collections import Counter
```

```
PASSAGE = (
    "It is a truth universally acknowledged, that a single man in possession "
    "of a good fortune, must be in want of a wife. However little known the "
    "feelings or views of such a man may be on his first entering a "
    "neighbourhood, this truth is so well fixed in the minds of the "
    "surrounding families, that he is considered the rightful property of "
    "some one or other of their daughters."
)

# Second attempt: lower-case, then keep runs of letters and apostrophes.
tokens = re.findall(r"[a-z']+", PASSAGE.lower())
counts = Counter(tokens)
print("tokens:", len(tokens), "| types:", len(counts))
for word, n in counts.most_common(8):
    print(f"{word:>8} {n}")
# tokens: 70 | types: 48
#        a 6
#       of 6
#      the 4
#       is 3
#       in 3
#    truth 2
#     that 2
#      man 2
```

Now the passage has 70 tokens and 48 types. The top of the list holds small grammatical words (`a`, `of`, `the`, `is`, `in`), not the interesting words, and that pattern is general. `a` and `of` tie at six, and `Counter.most_common` lists ties in the order they first appeared (an implementation detail worth not relying on). The next chapter, Words by the Numbers 2, asks how steeply these counts fall as you go down the list.

## Choices in Tokenising

Lower-casing merges “The” at the start of a sentence with “the” in the middle, which is usually what we want, but it also merges “Polish” with “polish”. Keeping apostrophes keeps `it’s` as one token but makes `don’t` and `dont` different. Every rule has a price.

| CHOICE | HELPS WITH | COSTS |
|---|---|---|
| Split on spaces | simplicity | punctuation sticks to words |

| CHOICE | HELPS WITH | COSTS |
|---|---|---|
| Lower-case | merging sentence-start words | merges names and common words |
| Letters only | clean words | splits `e-mail`, drops `3.50` |
| Keep apostrophes | `don't`, `it's` stay whole | quote marks can be mistaken for apostrophes |

## Measuring the Vocabulary

With tokens and types in hand, a few simple numbers describe a text. The **type–token ratio** is types divided by tokens: 48 ÷ 70 gives 0.686. A word seen only once is called a **hapax legomenon**, from Greek for "said once" (plural: hapax legomena).

```
import re
from collections import Counter
PASSAGE = (
    "It is a truth universally acknowledged, that a single man in possession "
    "of a good fortune, must be in want of a wife. However little known the "
    "feelings or views of such a man may be on his first entering a "
    "neighbourhood, this truth is so well fixed in the minds of the "
    "surrounding families, that he is considered the rightful property of "
    "some one or other of their daughters."
)

tokens = re.findall(r"[a-z']+", PASSAGE.lower())
counts = Counter(tokens)
total = len(tokens)
print("type-token ratio:", round(len(counts) / total, 3))
hapaxes = [w for w, n in counts.items() if n == 1]
print("words seen once:", len(hapaxes), f"({len(hapaxes) / len(counts):.0%} of types)")
print("share of the top 5 types:", f"{sum(n for _, n in counts.most_common(5)) / total:.
# type-token ratio: 0.686
# words seen once: 38 (79% of types)
# share of the top 5 types: 31.4%
```

Of the 48 types, 38 (79%) occur once, and the five commonest types account for 31.4% of all the tokens. Remember how small the sample is: with 70 tokens almost any word is a hapax, and the

type–token ratio tends to fall as a text gets longer, because common words keep repeating. So the ratio is not comparable between texts of different lengths. Words by the Numbers 6 returns to this when it compares authors.

## Counting Letters

The same method works one level down. Letters are tokens too, and in the passage the letter `e` leads comfortably.

```
from collections import Counter
PASSAGE = (
    "It is a truth universally acknowledged, that a single man in possession "
    "of a good fortune, must be in want of a wife. However little known the "
    "feelings or views of such a man may be on his first entering a "
    "neighbourhood, this truth is so well fixed in the minds of the "
    "surrounding families, that he is considered the rightful property of "
    "some one or other of their daughters."
)

letters = [ch for ch in PASSAGE.lower() if ch.isalpha()]
counts = Counter(letters)
total = len(letters)
print("letters:", total)
for ch, n in counts.most_common(5):
    print(f"{ch}: {n:3d}  {n / total:.1%}  " + "#" * n)
vowels = sum(counts[v] for v in "aeiou")
print(f"vowels: {vowels / total:.1%}")
# letters: 300
# e:  34  11.3%  ##################################
# i:  27  9.0%  ###########################
# o:  27  9.0%  ###########################
# t:  26  8.7%  ##########################
# s:  23  7.7%  #######################
# vowels: 38.3%
```

Of 300 letters, `e` has 34 (11.3%), and vowels make up 38.3%. That one sample is far too small to quote as the share of `e` in English; a larger reference count would be needed (check). Letter profiles are the subject of Words by the Numbers 3, where they tell languages apart.

## A Small Reusable Counter

The pieces so far fit in two functions that later chapters build on. Each snippet in the course imports what it uses, so you can paste any block into a fresh file and run it.

```
import re
from collections import Counter

def tokenise(text):
    # Lower-case, then keep runs of letters and apostrophes.
    return re.findall(r"[a-z']+", text.lower())

def word_report(text, top=5):
    tokens = tokenise(text)
    counts = Counter(tokens)
    return {
        "tokens": len(tokens),
        "types": len(counts),
        "top": counts.most_common(top),
    }

sample = "the cat sat on the mat and the dog sat on the cat"
print(word_report(sample, top=3))
# {'tokens': 13, 'types': 7, 'top': [('the', 4), ('cat', 2), ('sat', 2)]}
```

Counting is much older than code. The Dominican friar Hugh of Saint-Cher is generally credited with the first verbal concordance of the Bible, an index of where each word occurs, produced in Paris in the 13th century with many helpers. Sources disagree on the date: one puts completion at 1230 and another gives 1230 to 1239 (conflict; check). It is the same job our counter does, with parchment instead of a `Counter`.

> Common mistakes
> Counting `word` and `word,` as different things because punctuation was left in; forgetting to lower-case, so `The` and `the` are separate types; mixing up tokens and types (“how many words?” has two honest answers); comparing the type–token ratio of a short text with that of a long one; and quoting results from a tiny sample, such as this chapter’s 70-word passage, as if they were facts about a language.

### EXERCISE 1

Take a short invented passage of your own (the solution uses 28 words). Report the number of tokens and types, the five commonest words, and the list of words that occur exactly once.

View solution

### EXERCISE 2

Cut the sentence “Don’t pay $3.50 for a well-known e-mail tool; it’s cheaper in Sao Paulo.” into words three ways: splitting on spaces, keeping letters and apostrophes, and a pattern that also keeps hyphens, decimal points and digits. Compare the counts. Which result would you trust for a word count, and which for a price search?

View solution

### EXERCISE 3

Compare the letter profiles of the Austen sentence and a short invented passage: the commonest letter and its share, the share of vowels, and how many different letters appear. Do the two profiles look alike, and why is it risky to conclude anything from them?

View solution

### QUICK REFERENCE

- **Token:** one occurrence of a word; **type:** a distinct word; a text has at least as many tokens as types
- **Tokenising:** choosing where words start and end; every rule (case, punctuation, apostrophes, hyphens) has a cost
- **Type–token ratio:** types ÷ tokens; it tends to fall as texts get longer, so compare equal lengths only
- **Hapax:** a type seen once; in short texts most types are hapaxes
- **Python habit:** `re.findall` to tokenise, `Counter` to tally, `most_common(n)` for the top of the list
- **Check before trusting a count:** which tokeniser, how long the text is, and whether the sample is representative

**COMING NEXT**

Words by the Numbers 2 is about Zipf's law: rank and frequency, the long tail of rare words, log-log curves and testing the law on different texts.

# Words by the Numbers

Chapter 2 · Zipf's Law

Words by the Numbers 1 ended with a hint: in a tiny passage the commonest words were small grammatical ones, and most other words appeared once. This chapter asks how steep that drop is. The surprise, noticed by the American linguist George Kingsley Zipf, is that the drop follows a simple pattern across very different texts, and a few lines of arithmetic and Python are enough to see it, measure it and, importantly, to see where it stops working. The maths is light: division, a logarithm, and a line of best fit that Python computes for you.

**WHAT WAS RUN, AND WHAT TO TRUST**

Every Python snippet in this chapter and in the three exercise solutions was executed (Python 3.14), and the output in the comments is what Python printed. The real text used for counting is the reference documentation that ships inside Python itself (the `pydoc_data.topics` module): about 75,000 tokens of technical English, available on any machine, but not typical English prose, and its counts change a little between Python versions. Background facts come from web searches that returned summaries of encyclopedia and paper pages; such fetched pages are often truncated or model-summarised, so treat the wording as a paraphrase. The sources agree on the basic statement and the Brown Corpus figure but differ on exact numbers and framing (conflict; check). The vocabulary size and text length in the first snippet are *assumed, illustrative* numbers. Statements marked (check) were not confirmed against a source.

## Rank and Frequency

Sort the types of a text by how often they occur: the commonest gets **rank** 1, the next rank 2, and so on. Zipf's law, as the sources summarise it, says that the frequency of a word is roughly inversely proportional to its rank. The word of rank 2 should occur about half as often as the word of rank 1, rank 3 about a third as often, rank 10 about a tenth. Written as a formula, with $f$ the frequency and $r$ the rank, $f \approx C \div r$, where $C$ is a constant for the text. Another way to say it: rank × frequency stays roughly constant. Zipf set out the idea at length in his 1949 book on the "principle of least effort", which argued that speakers and listeners both try to save effort (this is the reference accounts' summary; check against the book).

Here is what a perfect 1/rank text would look like. The snippet assumes a vocabulary of 1,000 types and 100,000 tokens; both numbers are illustrative, chosen only to make the arithmetic

easy to follow.

```
# Ideal Zipf: the word of rank r has frequency proportional to 1/r.
VOCAB = 1000      # assumed vocabulary size (illustrative)
TOKENS = 100000   # assumed text length (illustrative)

norm = sum(1 / r for r in range(1, VOCAB + 1))
print("normaliser (harmonic sum):", round(norm, 3))
print("rank  expected count  rank*count")
for r in (1, 2, 3, 5, 10, 100, 1000):
    expected = TOKENS * (1 / r) / norm
    print(f"{r:4d}  {expected:14.1f}  {r * expected:10.1f}")
print("share of rank 1:", f"{(1 / norm):.1%}")
# normaliser (harmonic sum): 7.485
# rank  expected count  rank*count
#    1         13359.2     13359.2
#    2          6679.6     13359.2
#    3          4453.1     13359.2
#    5          2671.8     13359.2
#   10          1335.9     13359.2
#  100           133.6     13359.2
# 1000            13.4     13359.2
# share of rank 1: 13.4%
```

The last column never changes: that is the whole law. The constant 13,359 is simply the number of tokens divided by the harmonic sum 1 + 1/2 + 1/3 + … + 1/1000 (about 7.485), so rank 1 takes 13.4% of the text. Real texts only approximately follow this, as the next section shows.

## Testing It on Real Text

Now the same table for real text. The snippet tokenises the whole Python reference documentation with the tokeniser from Words by the Numbers 1 and prints the top ten ranks with their rank × count products.

```
import re
from collections import Counter
from pydoc_data.topics import topics
```

```
def tokenise(text):
    return re.findall(r"[a-z']+", text.lower())

text = " ".join(topics.values())
tokens = tokenise(text)
counts = Counter(tokens)
print("tokens:", len(tokens), "| types:", len(counts))
print("rank word        count  rank*count")
for rank, (word, n) in enumerate(counts.most_common(10), start=1):
    print(f"{rank:4d} {word:<10} {n:7d} {rank * n:11d}")
once = sum(1 for n in counts.values() if n == 1)
print("types seen once:", once, f"({once / len(counts):.0%} of types)")
# tokens: 75022 | types: 3445
# rank word        count  rank*count
#    1 the          5201        5201
#    2 a            2188        4376
#    3 is           2019        6057
#    4 of           1561        6244
#    5 in           1557        7785
#    6 to           1445        8670
#    7 and          1183        8281
#    8 for           945        7560
#    9 if            804        7236
#   10 are           760        7600
# types seen once: 918 (27% of types)
```

The top word, `the`, takes 5,201 of 75,022 tokens, which is 6.9%. For comparison, the sources report that in the Brown Corpus, a one-million-word sample of American English from the 1960s, `the` is also the commonest word and accounts for nearly 7% of all occurrences (69,971 out of slightly over a million; second-hand figures, check). The rank × count column wobbles between about 4,400 and 8,700 in the top ten, so the product is not constant, but it stays within a factor of two, over a range of counts from 5,201 down to 760. That is why people say "Zipf's law" for something this loose. Note also the bottom line of the output: 918 of the 3,445 types (27%) occur exactly once. That crowd of rare words is the **long tail**.

| PART OF THE CURVE | RANKS (THIS TEXT) | WHAT LIVES THERE |
|---|---|---|
| Head | 1 to about 10 | grammar words: `the`, `a`, `is`, `of` |
| Body | about 10 to several hundred | common content words and technical terms |

| PART OF THE CURVE | RANKS (THIS TEXT) | WHAT LIVES THERE |
|---|---|---|
| Long tail | hundreds to 3,445 | rare words; 918 appear once |

## Log-Log Plots and the Slope

If $f = C \div r$, take logarithms: $\log f = \log C - \log r$. So on a plot with log rank across and log frequency up, a Zipf text is a straight line with slope −1. A **log-log plot** turns a curve that plunges and then crawls along the floor into something the eye can judge. Python can fit the line without a plotting library. The snippet below uses the standard least-squares formula, then draws a crude text-mode picture in which each bar's length is ten times the base-10 logarithm of the count.

```
import math
import re
from collections import Counter
from pydoc_data.topics import topics

def tokenise(text):
    return re.findall(r"[a-z']+", text.lower())

def zipf_slope(tokens, max_rank=1000):
    # Least-squares line through (log rank, log count); returns the slope.
    freqs = sorted(Counter(tokens).values(), reverse=True)[:max_rank]
    xs = [math.log(r) for r in range(1, len(freqs) + 1)]
    ys = [math.log(f) for f in freqs]
    mx, my = sum(xs) / len(xs), sum(ys) / len(ys)
    num = sum((x - mx) * (y - my) for x, y in zip(xs, ys))
    den = sum((x - mx) ** 2 for x in xs)
    return num / den

tokens = tokenise(" ".join(topics.values()))
print("slope, top 1000 ranks:", round(zipf_slope(tokens), 2))
print("slope, top 100 ranks: ", round(zipf_slope(tokens, 100), 2))

# A crude log-log picture: bar length = log10(count) * 10.
counts = sorted(Counter(tokens).values(), reverse=True)
for rank in (1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024):
    print(f"{rank:5d} {counts[rank - 1]:6d} " + "#" * round(math.log10(counts[rank - 1])
# slope, top 1000 ranks: -1.04
```

```
# slope, top 100 ranks:  -0.81
#     1   5201 ####################################
#     2   2188 ###############################
#     4   1561 ######################################
#     8    945 ##############################
#    16    639 ############################
#    32    313 ########################
#    64    177 ####################
#   128     95 ###################
#   256     52 ################
#   512     22 ############
#  1024      8 #########
```

The slope over the top 1,000 ranks is −1.04, close to the textbook −1, while over only the top 100 ranks it is −0.81. The same text gives different slopes depending on which ranks you fit, so a slope is always quoted together with the rank range. In the picture, each doubling of rank takes the count down by a factor that is not quite two at the left (5,201, 2,188, 1,561) and somewhat steeper later (95, 52 at ranks 128 and 256), which is a straight-ish line on log axes with a gentle bend. The usual fix in the literature is the Zipf–Mandelbrot form, which has frequency proportional to $1/(\text{rank} + b)^a$, with two adjustable numbers instead of one; the sources attribute it to Benoit Mandelbrot (check the details).

| FORM | FREQUENCY AT RANK *R* | STRAIGHT LINE ON LOG-LOG? |
|---|---|---|
| Zipf (simple) | $C \div r$ | yes, slope −1 |
| Generalised | $C \div r^a$ | yes, slope $-a$ |
| Zipf–Mandelbrot | $C \div (r + b)^a$ | bends at low ranks, straight later |

## Does Text Length Matter, and Is It Special to Language?

Two honest tests. First, fit the same text at different lengths, and add a second, quite different text: the doc-strings of Python's built-in names (a few thousand tokens).

```
import builtins
import inspect
```

```
import math
import re
from collections import Counter
from pydoc_data.topics import topics

def tokenise(text):
    return re.findall(r"[a-z']+", text.lower())

def zipf_slope(tokens, max_rank=1000):
    freqs = sorted(Counter(tokens).values(), reverse=True)[:max_rank]
    xs = [math.log(r) for r in range(1, len(freqs) + 1)]
    ys = [math.log(f) for f in freqs]
    mx, my = sum(xs) / len(xs), sum(ys) / len(ys)
    num = sum((x - mx) * (y - my) for x, y in zip(xs, ys))
    den = sum((x - mx) ** 2 for x in xs)
    return num / den

docs = tokenise(" ".join(topics.values()))
# Slope against text length: the same text, first N tokens.
for n in (1000, 10000, 40000, len(docs)):
    print(f"first {n:7d} tokens: slope {zipf_slope(docs[:n], 1000):.2f}  types {len(set(

# A second, different text: the docstrings of the built-in names.
doc_text = " ".join(inspect.getdoc(obj) or "" for obj in vars(builtins).values())
small = tokenise(doc_text)
print("builtin docstrings:", len(small), "tokens, slope", round(zipf_slope(small, 300), 
      "| top word:", Counter(small).most_common(1)[0])
# first    1000 tokens: slope -0.90  types 286
# first   10000 tokens: slope -1.15  types 1284
# first   40000 tokens: slope -1.06  types 2553
# first   75022 tokens: slope -1.04  types 3445
# builtin docstrings: 4908 tokens, slope -0.93 | top word: ('the', 342)
```

With only the first 1,000 tokens the slope is −0.90; at 10,000 it is −1.15; at 40,000, −1.06; and for the full text, −1.04. Short samples are noisy, so do not judge the law from a paragraph (the 70-word passage from Words by the Numbers 1 has far too few repeats). The small doc-string collection gives −0.93 over its top 300 ranks, and its top word is again `the`. These are two samples of one narrow genre; they say nothing about novels or speech.

The second test is a famous deflation. The psychologist George Miller pointed out in 1957 that text typed by a monkey hitting random keys, including the space bar, also yields a power law;

Wentian Li gave a formal argument in 1992 (the sources agree on this core story, but differ on how to credit Miller, Mandelbrot and Li; conflict; check). The snippet below does the experiment with a fixed random seed: one million random characters from 26 letters and the space, all equally likely.

```
import math
import random
import re
from collections import Counter
from pydoc_data.topics import topics

def tokenise(text):
    return re.findall(r"[a-z']+", text.lower())

def zipf_slope(tokens, max_rank=1000):
    freqs = sorted(Counter(tokens).values(), reverse=True)[:max_rank]
    xs = [math.log(r) for r in range(1, len(freqs) + 1)]
    ys = [math.log(f) for f in freqs]
    mx, my = sum(xs) / len(xs), sum(ys) / len(ys)
    num = sum((x - mx) * (y - my) for x, y in zip(xs, ys))
    den = sum((x - mx) ** 2 for x in xs)
    return num / den

# Monkey typing: 26 letters and a space, all equally likely, fixed seed.
random.seed(1)
typed = "".join(random.choice("abcdefghijklmnopqrstuvwxyz ") for _ in range(1_000_000))
monkey = typed.split()
real = tokenise(" ".join(topics.values()))
for name, toks in (("monkey", monkey), ("real text", real)):
    print(f"{name:10} tokens {len(toks):7d}  types {len(set(toks)):6d}  slope {zipf_slop
print("most common monkey 'words':", Counter(monkey).most_common(4))
# monkey     tokens   35637  types  33591  slope -0.73
# real text  tokens   75022  types   3445  slope -1.04
# most common monkey 'words': [('l', 64), ('a', 61), ('m', 61), ('f', 58)]
```

The monkey text has 35,637 “words” of which 33,591 are different, and the fit over the top 1,000 ranks gives −0.73, not −1. Its curve is a staircase, because all words of the same length are equally likely, so many types tie on the same count. Short words are common simply because there are fewer of them to type. The lesson is not that the law is meaningless but that a straight

line on log-log axes is a weak test: judge a claim by its slope, its rank range and how much data lies behind it, and ask what a random-text control gives.

| TEXT | TOKENS | TOP-1000 SLOPE |
|---|---|---|
| Python documentation | 75,022 | −1.04 |
| Monkey typing (seed 1) | 35,637 | −0.73 |
| Built-in doc-strings | 4,908 | −0.93 (top 300) |

Common mistakes

Calling a rank–frequency pattern “proof” of the law from a very short text; quoting a slope without saying which ranks were fitted; plotting frequency against rank on ordinary axes, where the tail is invisible; forgetting that ties at the tail make steps; treating the two samples here as typical English; and reading the “constant” rank × frequency as exact when it only holds within a factor of about two.

**EXERCISE 1**

Build a rank–frequency table for a short invented passage and print rank × count for the top eight ranks. How many types share the lowest count, and what does that do to the shape of the tail?

View solution

**EXERCISE 2**

Predict the counts at ranks 1, 2, 5, 10, 50, 100, 500 and 1,000 with the ideal 1/rank formula, using the real text’s own token and type totals, and print the ratio of actual to predicted. Where does the simple law over-predict, and where under-predict?

View solution

**EXERCISE 3**

Fit the log-log slope separately over the rank bands 1–10, 10–100, 100–1,000 and 1,000–2,000, and count the types with frequency 1. In which band does a single straight line stop being a good description, and why?

View solution

### QUICK REFERENCE

- **Rank:** position in the list sorted by frequency, commonest = 1
- **Zipf's law:** frequency ≈ $C \div$ rank, so rank × frequency is roughly constant
- **Log-log plot:** log rank against log frequency; a straight line of slope −1 means Zipf
- **Long tail:** many rare words; in a 75,022-token text, 27% of the types occurred once
- **Zipf–Mandelbrot:** adds an offset $b$ to the rank so the head can bend
- **Random-text control:** monkey typing also gives a rough power law, so check slope, range and data size
- **Python habit:** `sorted(Counter(tokens).values(), reverse=True)` gives the frequencies by rank

### COMING NEXT

Words by the Numbers 3 is about letters and secrets: letter frequency in languages, early frequency analysis and telling languages apart by their letter profiles.

# Words by the Numbers

Chapter 3 · Letters and Secrets

Words by the Numbers 2 counted whole words and found a steep, lopsided pattern. This chapter drops down a level and counts letters. Letters are far fewer than words (26 of them, in the English alphabet) so even a short text gives a usable count, and the counts turn out to be stubbornly regular. That regularity has two famous uses: breaking secret messages, which is where this story starts, and recognising which language a text is written in without understanding a word of it. The maths is light: percentages, a remainder after division, and a distance that is just a sum of differences.

> **WHAT WAS RUN, AND WHAT TO TRUST**
>
> Every Python snippet in this chapter and in the three exercise solutions was executed (Python 3.14), and the output in the comments is what Python printed. The English text counted in the first snippet is the reference documentation that ships inside Python (the `pydoc_data.topics` module), about 363,000 letters of technical English; it is not typical prose, and its counts change a little between Python versions. The short passages used for the language-recognition snippet were *written for this course* (the same small story in five languages, with my own translations, not checked by native speakers), so treat that experiment as a demonstration on tiny, parallel, same-topic samples and not as a measurement of the languages. Background facts come from web searches that returned summaries of web pages; such fetched pages are often truncated or model-summarised, so treat the wording as a paraphrase. The sources agree on Al-Kindi's treatise and its method but disagree on the date of Babbage's work on the Vigenère cipher (conflict, below). Messages and shift keys in the cipher snippets are *invented, illustrative* examples. Statements marked (check) were not confirmed against a source.

## Letters in Real Text

Start by counting. The snippet below keeps only the 26 unaccented letters, lowercased, from the whole Python reference documentation and prints the eight commonest. If every letter were equally likely each would take 1 ÷ 26, about 3.8%. Real English is nothing like that.

```
import re
from collections import Counter
from pydoc_data.topics import topics
```

```
def letters_only(text):
    return re.findall(r"[a-z]", text.lower())

letters = letters_only(" ".join(topics.values()))
counts = Counter(letters)
total = len(letters)
print("letters:", total, "| an even spread would give each", f"{1 / 26:.1%}")
print("letter   count  share")
for letter, n in counts.most_common(8):
    print(f"{letter:>6} {n:7d} {n / total:6.1%}")
rarest = sorted(counts.items(), key=lambda pair: pair[1])[:4]
print("rarest:", [(letter, f"{n / total:.2%}") for letter, n in rarest])
# letters: 363372 | an even spread would give each 3.8%
# letter   count  share
#      e   48945  13.5%
#      t   37536  10.3%
#      a   28415   7.8%
#      i   26992   7.4%
#      n   26949   7.4%
#      s   26622   7.3%
#      o   23879   6.6%
#      r   21733   6.0%
# rarest: [('z', '0.09%'), ('q', '0.19%'), ('j', '0.32%'), ('k', '0.38%')]
```

The letter `e` takes 13.5% of all letters, more than three times the even-spread share, and the top four letters (e, t, a, i) together make up about 39%. At the other end, `z` accounts for under a tenth of one percent. The familiar "e, t, a..." order is the standard story about English; here the order is e, t, a, i, n, s, o, r, because this is documentation text with plenty of *i* from words such as *is*, *if* and *in*. Different texts give slightly different shares, but a text of any decent length reorders only the middle of the list. One source in my search also notes that `th` is the most likely pair of letters in English; pairs are the subject of Words by the Numbers 4, so take that as a preview, not a result.

| UNIT COUNTED | TYPES TO COUNT | TEXT NEEDED FOR A STABLE PICTURE |
|---|---|---|
| Letters | 26 | a few dozen letters gave usable results in the experiment below |

| UNIT COUNTED | TYPES TO COUNT | TEXT NEEDED FOR A STABLE PICTURE |
|---|---|---|
| Words | thousands, with a long tail (Words by the Numbers 2) | tens of thousands of tokens for a steady slope |

## Al-Kindi and the Shift Cipher

Regular letter counts are a gift to anyone trying to read a message not meant for them. According to the sources I found, the earliest known description of breaking a cipher by counting letters is a treatise by the Arab scholar Al-Kindi, *A Manuscript on Deciphering Cryptographic Messages*, usually dated to about 850 CE in Baghdad, with a copy rediscovered in an Ottoman archive in Istanbul in 1987 (a Simon Singh article and a history-of-information page; second-hand, check). The method is simple: count how often each symbol appears in the ciphertext, then match the commonest symbols to the commonest letters of the language. One summary says he began from the observation that in a given language the letters settle at stable frequencies. Another source cautions that the manuscript's examples are easiest to explain with the English alphabet, but that Al-Kindi worked on Arabic.

The simplest cipher to attack this way is the **shift cipher**, usually called the Caesar cipher after the tradition that Julius Caesar used a shift of three (my recollection of the Suetonius account; not confirmed in my search; check). Each letter moves a fixed number of places along the alphabet, wrapping round from `z` to `a`. Wrapping is where a little arithmetic comes in: the new position is (old position + *k*) remainder 26, which Python writes `(i + k) % 26`. There are only 26 possible shifts, so you could try them all, but frequency gives a shortcut: assume the commonest ciphertext letter stands for `e`, and the shift is the distance between them.

```
import string
from collections import Counter

ALPHA = string.ascii_lowercase

def shift(text, k):
    # Move every letter k places along the alphabet; leave other characters alone.
    return "".join(
        ALPHA[(ALPHA.index(ch) + k) % 26] if ch in ALPHA else ch for ch in text
    )
```

```
plain = ("meet me by the old bridge when the evening bells are heard and "
         "bring the letter that the secretary left on the table")
secret = shift(plain, 3)
print("ciphertext:", secret)

cipher_counts = Counter(ch for ch in secret if ch in ALPHA)
commonest = cipher_counts.most_common(1)[0][0]
guess = (ALPHA.index(commonest) - ALPHA.index("e")) % 26
print("commonest cipher letter:", commonest, "| assumed to be e, so shift =", guess)
print("recovered:", shift(secret, -guess))
# ciphertext: phhw ph eb wkh rog eulgjh zkhq wkh hyhqlqj ehoov duh khdug dqg eulqj wkh ol
# commonest cipher letter: h | assumed to be e, so shift = 3
# recovered: meet me by the old bridge when the evening bells are heard and bring the le
```

The commonest cipher letter is `h`, which is three places after `e`, so the shift is 3 and the message comes back intact. The message here is 116 characters long, invented for the example, and plenty of `e`s made the guess easy. The next section shows what happens when it is not.

## When the “e” Guess Fails

Frequency analysis is a statistical argument, and statistics need data. A very short message may have no `e` at all. Here are two ways to attack the same ciphertexts: the “commonest letter is e”

shares are closest to the English reference profile (using the sum of absolute differences, so the distance is 0 for identical profiles). The reference profile comes from the documentation text counted earlier. All four messages are enciphered with the same invented key, 7.

```
import string
from collections import Counter
from pydoc_data.topics import topics

ALPHA = string.ascii_lowercase

def shift(text, k):
    return "".join(
        ALPHA[(ALPHA.index(ch) + k) % 26] if ch in ALPHA else ch for ch in text
    )
```

```
def shares(text):
    letters = [ch for ch in text.lower() if ch in ALPHA]
    counts = Counter(letters)
    return [counts[ch] / len(letters) for ch in ALPHA]

english = shares(" ".join(topics.values()))   # reference profile

def crack_by_e(cipher):
    commonest = Counter(ch for ch in cipher if ch in ALPHA).most_common(1)[0][0]
    return (ALPHA.index(commonest) - ALPHA.index("e")) % 26

def crack_by_fit(cipher):
    # Try all 26 shifts; keep the one whose letter shares are closest to English.
    def gap(k):
        trial = shares(shift(cipher, -k))
        return sum(abs(a - b) for a, b in zip(trial, english))
    return min(range(26), key=gap)

messages = [
    "the secret meeting is moved to the usual place after the evening bells",
    "attack at dawn",
    "buy milk and a loaf of bread",
    "send help to the old mill",
]
KEY = 7
for plain in messages:
    cipher = shift(plain, KEY)
    by_e, by_fit = crack_by_e(cipher), crack_by_fit(cipher)
    print(f"{len(plain):3d} chars | e-guess {'ok  ' if by_e == KEY else 'FAIL'}"
          f" | best-fit {'ok  ' if by_fit == KEY else 'FAIL'} | {plain[:28]}")
#  70 chars | e-guess ok   | best-fit ok   | the secret meeting is moved
#  14 chars | e-guess FAIL | best-fit FAIL | attack at dawn
#  28 chars | e-guess FAIL | best-fit FAIL | buy milk and a loaf of bread
#  25 chars | e-guess FAIL | best-fit ok   | send help to the old mill
```

On the 70-character message both rules succeed. On `attack at dawn` (14 characters) and `buy milk and a loaf of bread` (28 characters) both fail; on `send help to the old mill` (25 characters) only best-fit succeeds. Four messages prove nothing statistical, but they show the shape of the problem: with a few dozen letters the counts are noisy, and a smarter rule, or a human who notices that the output is gibberish, has to take over. Note, too, that the reference

profile here comes from technical documentation, not from the kind of message a spy would write, so a profile that matches the sender's style would do better.

| CIPHER | KEYS | LETTER COUNTS OF THE CIPHERTEXT |
|---|---|---|
| Shift (Caesar) | 26 shifts, including the do-nothing shift of 0 | same shape as the plaintext, only moved along the alphabet |
| Simple substitution | 26! = 403,291,461,126,605,635,584,000,000 (about $4.0 \times 10^{26}$, computed by Python) | same set of counts, each tied to one symbol: Al-Kindi's method still applies |
| Vigenère | a keyword; each letter is shifted by the matching keyword letter in turn | flattened, because one plaintext letter becomes several different symbols |

That last row is why the Vigenère cipher was thought unbreakable for a long time. A search on the Kasiski examination reports that Friedrich Kasiski published a method in 1863, based on spotting repeated strings in the ciphertext that reveal the keyword length, and that Charles Babbage had independently found a way into the cipher earlier without ever explaining it. The sources conflict on when: one gives 1846, another 1854 (conflict; check). Once the length is known, each position is a separate shift cipher and the e-guess works again.

## Letter Profiles: Telling Languages Apart

Turn the idea around. Instead of using a known language to read a hidden message, use the letter counts to identify the language. A **profile** is a list of 26 numbers: the share of each letter. Two texts in the same language should have close profiles, and two different languages should differ. To compare profiles use the distance from the last section: add up the absolute differences, letter by letter. The smaller the sum, the more alike the texts.

The experiment below uses five short training passages (the same invented harbour story in English, French, German, Spanish and Italian), builds a profile from each, and then tries to say which language each of five different test sentences is in. Accents are stripped first (`é` becomes `e`, `ö` becomes `o`), which keeps the profile to 26 letters. To see how much text is needed, it classifies the first 20, 40 and 80 letters of each test sentence, and then all of it. The passages are long because the data is in the snippet, so every line can be run as it stands.

```
TRAIN = {
    "English": (
        "The old harbour was quiet in the early morning, and the fishermen were "
        "already mending their nets on the stone quay. A cold wind came in from "
        "the sea, carrying the smell of salt and tar, while the children of the "
        "village walked slowly to school with their books under their arms. "
        "Nobody spoke much, for the week had been long and the weather had been "
        "hard on everyone who worked on the water."
    ),
    "French": (
        "Le vieux port était calme au petit matin, et les pêcheurs réparaient "
        "déjà leurs filets sur le quai de pierre. Un vent froid venait de la mer, "
        "apportant l'odeur du sel et du goudron, pendant que les enfants du "
        "village marchaient lentement vers l'école, leurs livres sous le bras. "
        "Personne ne parlait beaucoup, car la semaine avait été longue et le "
        "temps avait été dur pour tous ceux qui travaillaient sur l'eau."
    ),
    "German": (
        "Der alte Hafen war am frühen Morgen still, und die Fischer flickten "
        "schon ihre Netze am steinernen Kai. Ein kalter Wind kam vom Meer und "
        "brachte den Geruch von Salz und Teer mit, während die Kinder des Dorfes "
        "langsam zur Schule gingen, ihre Bücher unter dem Arm. Niemand sprach "
        "viel, denn die Woche war lang gewesen und das Wetter hatte allen "
        "zugesetzt, die auf dem Wasser arbeiteten."
    ),
    "Spanish": (
        "El viejo puerto estaba tranquilo a primera hora de la mañana, y los "
        "pescadores ya remendaban sus redes en el muelle de piedra. Un viento "
        "frío llegaba del mar, trayendo el olor a sal y alquitrán, mientras los "
        "niños del pueblo caminaban despacio hacia la escuela con los libros "
        "bajo el brazo. Nadie hablaba mucho, porque la semana había sido larga "
        "y el tiempo había sido duro para todos los que trabajaban en el agua."
    ),
    "Italian": (
        "Il vecchio porto era tranquillo di primo mattino, e i pescatori "
        "rammendavano già le loro reti sulla banchina di pietra. Un vento freddo "
        "veniva dal mare, portando l'odore di sale e di catrame, mentre i "
        "bambini del villaggio camminavano lentamente verso la scuola con i "
        "libri sotto il braccio. Nessuno parlava molto, perché la settimana era "
        "stata lunga e il tempo era stato duro per tutti quelli che lavoravano "
        "sull'acqua."
    ),
}
```

```
TEST = {
    "English": (
        "She opened the window and listened to the rain falling on the roof of "
        "the little house, thinking about the letter that had arrived the day "
        "before and what she should write in answer to it."
    ),
    "French": (
        "Elle ouvrit la fenêtre et écouta la pluie qui tombait sur le toit de la "
        "petite maison, en pensant à la lettre qui était arrivée la veille et à "
        "ce qu'elle devait écrire en réponse."
    ),
    "German": (
        "Sie öffnete das Fenster und hörte dem Regen zu, der auf das Dach des "
        "kleinen Hauses fiel, und dachte an den Brief, der am Tag zuvor "
        "angekommen war, und was sie darauf antworten sollte."
    ),
    "Spanish": (
        "Ella abrió la ventana y escuchó la lluvia que caía sobre el tejado de "
        "la casita, pensando en la carta que había llegado el día anterior y en "
        "lo que debía escribir como respuesta."
    ),
    "Italian": (
        "Lei aprì la finestra e ascoltò la pioggia che cadeva sul tetto della "
        "piccola casa, pensando alla lettera che era arrivata il giorno prima e "
        "a cosa dovesse scrivere in risposta."
    ),
}

import string
import unicodedata
from collections import Counter

ALPHA = string.ascii_lowercase

def base_letters(text):
    # Strip accents (e-acute becomes e, o-umlaut becomes o) and keep a-z only.
    decomposed = unicodedata.normalize("NFD", text.lower())
    return [ch for ch in decomposed if ch in ALPHA]

def profile(text):
    letters = base_letters(text)
    counts = Counter(letters)
    return [counts[ch] / len(letters) for ch in ALPHA]

def distance(p, q):
```

```
    # Sum of the absolute differences between two letter-share lists.
    return sum(abs(a - b) for a, b in zip(p, q))

profiles = {lang: profile(text) for lang, text in TRAIN.items()}

def classify(text):
    p = profile(text)
    return min(profiles, key=lambda lang: distance(p, profiles[lang]))

for lang, p in profiles.items():
    top = sorted(zip(ALPHA, p), key=lambda pair: -pair[1])[:3]
    print(f"{lang:8}", ", ".join(f"{letter} {share:.1%}" for letter, share in top))

for n in (20, 40, 80, None):
    wrong = []
    for lang, text in TEST.items():
        sample = "".join(base_letters(text)[:n])
        if classify(sample) != lang:
            wrong.append((lang, classify(sample)))
    label = "all" if n is None else n
    print(f"first {label} letters: {5 - len(wrong)}/5 right", wrong)
# English  e 14.7%, h 8.3%, o 8.0%
# French   e 17.4%, a 9.8%, t 8.8%
# German   e 17.4%, n 10.0%, r 8.1%
# Spanish  a 16.5%, e 11.4%, o 8.7%
# Italian  a 13.2%, e 9.9%, o 9.4%
# first 20 letters: 4/5 right [('Italian', 'French')]
# first 40 letters: 5/5 right []
# first 80 letters: 5/5 right []
# first all letters: 5/5 right []
```

The top-three lists already look different: German and French put `e` at 17.4%, while Spanish and Italian lead with `a`. English has `h` as its second letter in this small sample, from words like *the*. The classifier gets all five right given 40 letters or more, and four of five at 20 letters, where an Italian sentence is taken for French. That is a nice result, but it is a very small, friendly test: the passages are translations of each other, there are five sentences, and published frequency tables, built from far larger collections, would differ from these small-sample shares (the 14.7% for `e` in English here, for instance, is well above the figure I recall for large English corpora; check). Real language identifiers use more than single letters, typically groups of two or three, which is where the next chapter goes.

| SAMPLE OF THE TEST SENTENCE | CORRECT OF 5 | MISTAKE |
|---|---|---|
| first 20 letters | 4 | Italian taken for French |
| first 40 letters | 5 | none |
| first 80 letters | 5 | none |
| whole sentence | 5 | none |

Common mistakes

Assuming the commonest ciphertext letter is always `e` (short messages, or messages with unusual vocabulary, break this); building a profile from a text in a different genre and expecting a perfect match; forgetting to strip accents or to lowercase, so that `E` and `é` count as different letters; treating a result on five sentences as a measured accuracy; and forgetting that the shift formula needs `% 26` to wrap around the end of the alphabet.

### EXERCISE 1

Count the letters in a short invented passage of your own (two sentences will do). Print the top five letters with their percentage shares, the share of `e` compared with the even-spread 3.8%, and how many of the 26 letters never appear.

View solution

### EXERCISE 2

Encrypt five short messages with different shift keys. Break each with two rules: (A) the commonest letter is `e`; (B) try each of the three commonest letters as `e` and keep the shift that makes the words “the”, “and”, “to” appear most often. How many does each rule get right?

View solution

### EXERCISE 3

Using the five training passages from this chapter, print the three closest pairs of languages by profile distance and the furthest pair. Then add the Python documentation as a second kind of English and see whether it sits closer to the story English or to the other languages.

View solution

### QUICK REFERENCE

- **Letter profile:** a list of 26 shares, one per letter, summing to 100%
- **Even spread:** 1 ÷ 26, about 3.8% each; real English is far from it ( `e` was 13.5% in the documentation text)
- **Frequency analysis:** match the commonest ciphertext symbols to the commonest letters of the language (Al-Kindi, about 850 CE per the sources)
- **Shift cipher:** `ALPHA[(i + k) % 26]`; to undo it, shift by `-k`
- **Distance between profiles:** `sum(abs(a - b) for a, b in zip(p, q))`, smaller means more alike
- **Limits:** short messages, unusual vocabulary and genre differences all weaken the argument
- **Strip accents:** `unicodedata.normalize("NFD", text)`, then keep only `a` to `z`

### COMING NEXT

Words by the Numbers 4 is about neighbouring words: bigrams and trigrams, collocations, and which word pairs are more than chance.

# Words by the Numbers

Chapter 4 · Neighbouring Words

Words by the Numbers 3 ended by saying that real language identifiers look at groups of letters, not single ones. The same step works for words: stop counting words one at a time and count the pairs and triples that sit next to each other. Some pairs are just common words that happen to be common (*of the*); others are pairs that stick together far more than their parts would explain (*floating point*). Telling the two apart is the whole chapter. The maths is light: multiplication, division, a square root and a base-2 logarithm, all done by Python.

**WHAT WAS RUN, AND WHAT TO TRUST**

Every Python snippet in this chapter and in the three exercise solutions was executed (Python 3.14), and the output in the comments is what Python printed. The real text counted is the reference documentation that ships inside Python (the `pydoc_data.topics` module), about 74,500 word tokens of technical English; it is not typical prose, and its counts change a little between Python versions, so expect small differences on your machine. Word pairs never cross from one documentation topic into the next. The tiny sentences in the first snippet and in Exercise 1 are *invented, illustrative* examples. Background facts come from web searches that returned summaries of web pages; such fetched pages are often truncated or model-summarised, so treat the wording as a paraphrase. A search confirmed the details of the Church and Hanks paper and the low-frequency weakness of the association score used below; it did not return anything on Firth or Halliday, so those two attributions are my recollection and are marked (check). Statements marked (check) were not confirmed against a source.

## Pairs and Triples of Words

A **bigram** is two neighbouring words, a **trigram** is three, and the general name is **n-gram**. Making them takes one trick: line the list of words up against itself shifted by one place, and zip the two lists together. A text of $n$ words has $n - 1$ bigrams and $n - 2$ trigrams, because the last word has no right-hand neighbour.

```
from collections import Counter

text = "the cat sat on the mat and the cat saw the dog"
words = text.split()
bigrams = list(zip(words, words[1:]))
```

```
trigrams = list(zip(words, words[1:], words[2:]))
print(len(words), "words,", len(bigrams), "bigrams,", len(trigrams), "trigrams")
print(bigrams[:4])
print(trigrams[:3])
print(Counter(bigrams).most_common(2))
# 12 words, 11 bigrams, 10 trigrams
# [('the', 'cat'), ('cat', 'sat'), ('sat', 'on'), ('on', 'the')]
# [('the', 'cat', 'sat'), ('cat', 'sat', 'on'), ('sat', 'on', 'the')]
# [(('the', 'cat'), 2), (('cat', 'sat'), 1)]
```

The sentence has 12 words, so 11 bigrams and 10 trigrams. `the cat` is the commonest bigram, with a count of 2, because *the cat* turns up twice. Everything else about n-grams is this same idea scaled up, so the real question is what the counts tell you.

## The Commonest Pairs Are Dull

Run the same idea over the whole documentation text. The snippet builds word, bigram and trigram counts, prints how many different bigrams and trigrams there are, how many appear only once, and the top few of each.

```
import re
from collections import Counter
from pydoc_data.topics import topics

def tokens(text):
    return re.findall(r"[a-z]+(?:'[a-z]+)?", text.lower())

uni, bi, tri = Counter(), Counter(), Counter()
for doc in topics.values():          # n-grams never cross from one document to the next
    w = tokens(doc)
    uni.update(w)
    bi.update(zip(w, w[1:]))
    tri.update(zip(w, w[1:], w[2:]))
N = sum(uni.values())
print("tokens:", N, "| distinct bigrams:", len(bi), "| distinct trigrams:", len(tri))
print("bigrams seen once:", f"{sum(1 for c in bi.values() if c == 1) / len(bi):.0%}",
      "| trigrams seen once:", f"{sum(1 for c in tri.values() if c == 1) / len(tri):.0%}")
print("top bigrams: ", [(" ".join(g), c) for g, c in bi.most_common(8)])
print("top trigrams:", [(" ".join(g), c) for g, c in tri.most_common(6)])
```

```
# tokens: 74538 | distinct bigrams: 27520 | distinct trigrams: 46967
# bigrams seen once: 53% | trigrams seen once: 66%
# top bigrams:  [('of the', 532), ('in the', 447), ('to the', 225), ('if the', 209), ('i
# top trigrams: [('changed in version', 128), ('self other object', 80), ('added in vers
```

The top bigrams are the pairs you would predict from Words by the Numbers 2: small grammatical words next to each other (*of the*, *in the*, *to the*). The documentation also shows its own habits: *in version* and *changed in version* come from the repeated notes of the form "Changed in version 3.x" (the version text is my reading of the output; check). Notice the last point about sparsity: 53% of the bigrams, and 66% of the trigrams, occur exactly once. The more words you glue together, the more of the possible combinations you have never seen.

| UNIT | DISTINCT KINDS | SEEN ONLY ONCE |
|---|---|---|
| Words | thousands (Words by the Numbers 2) | a large share of the kinds (check) |
| Bigrams | 27,520 | 53% |
| Trigrams | 46,967 | 66% |

## Collocations

A **collocation** is a pair (or longer group) of words that go together more than their meanings alone would require. The idea has a famous slogan, usually credited to the British linguist J. R. Firth: you shall know a word by the company it keeps (my recollection; the exact wording and year not confirmed; check). A standard textbook illustration, usually credited to M. A. K. Halliday, is that English speakers say *strong tea* and *heavy rain*, but not *powerful tea* (also from memory; check). Nothing in the meaning of *strong* forbids *strong rain*; it is simply not what people say. Dictionary makers care about this, and in 1990 Kenneth Church and Patrick Hanks published a paper in *Computational Linguistics* (volume 16, pages 22 to 29, per the search result) proposing to estimate such word associations automatically from large text collections, using a measure based on mutual information. That measure is the one coming up.

## More Than Chance

To say a pair is "more than chance" you need a figure for chance. Suppose the words were shuffled into random order. Word *a* makes up a share *c*(a) ÷ *N* of the text and word *b* makes up *c*(b) ÷ *N*, so the pair *a b* should turn up about *c*(a) × *c*(b) ÷ *N* times, the **expected count**. Compare it with the count actually seen. The usual way to compare is a ratio on a log scale, **pointwise mutual information** (PMI): $\log_2$(seen ÷ expected). A PMI of 0 means chance, 1 means twice as often as chance, 2 four times, and so on.

```
import math, re
from collections import Counter
from pydoc_data.topics import topics

def tokens(text):
    return re.findall(r"[a-z]+(?:'[a-z]+)?", text.lower())

uni, bi = Counter(), Counter()
for doc in topics.values():
    w = tokens(doc)
    uni.update(w)
    bi.update(zip(w, w[1:]))
N = sum(uni.values())

def pmi(pair):
    a, b = pair
    expected = uni[a] * uni[b] / N        # pairs expected if words were shuffled
    return math.log2(bi[pair] / expected)

for pair in [("of", "the"), ("in", "the"), ("is", "a")]:
    a, b = pair
    print(f"{' '.join(pair):7} seen {bi[pair]:5d} | expected {uni[a] * uni[b] / N:7.1f}
for floor in (1, 5, 30):
    ranked = sorted((p for p in bi if bi[p] >= floor), key=pmi, reverse=True)[:5]
    print(f"min count {floor:2d}:", [(" ".join(p), bi[p], round(pmi(p), 1)) for p in ranl
# of the  seen   532 | expected   109.0 | PMI  2.29
# in the  seen   447 | expected   108.8 | PMI  2.04
# is a    seen   162 | expected    59.8 | PMI  1.44
# min count  1: [('generation phase', 1, 16.2), ('fstring tstring', 1, 16.2), ('pyarg pa
# min count  5: [('latin capital', 5, 13.9), ('post mortem', 7, 13.4), ('triple quoted',
# min count 30: [('floating point', 36, 9.8), ('e g', 51, 8.6), ('finally clause', 32, 8
```

Even *of the* beats chance: 532 seen against about 109 expected, a ratio near 4.9 (PMI 2.29). That makes sense, because English grammar does put *the* after *of* more often than a shuffle would. The more important lesson is in the last three lines. With no minimum count, the top five pairs all have a PMI of 16.2 and each was seen once, and that is not a coincidence: a pair seen once between two words that each appear once has PMI $\log_2(N)$, and $\log_2(74{,}538)$ is about 16.2.

Those are rare words, not collocations. Raising the minimum count to 5 gives *latin capital*, *post mortem* and *triple quoted*; at 30 it gives *floating point*, *finally clause* and *except clause*, which really are the kind of fixed phrase you hoped to find. The search I ran says the same thing about PMI in general: it tends to favour low-frequency events, so it is usually paired with a frequency threshold.

| MINIMUM COUNT | TOP PAIR FOUND | WHAT IT SHOWS |
|---|---|---|
| 1 | generation phase (seen once, PMI 16.2) | rare words sitting together by accident |
| 5 | latin capital (seen 5 times, PMI 13.9) | real phrases, but narrow ones |
| 30 | floating point (seen 36 times, PMI 9.8) | established technical phrases |

## Does a Shuffle Agree?

The expected count is an arithmetic shortcut. You can also test it directly: shuffle the words inside each documentation topic, count the pairs again, and see what a text without word order produces. If *floating point* really is a phrase, shuffled text should almost never produce it. The same snippet also computes a second score, the **t-score**: (seen − expected) ÷ √seen. It rewards pairs that are both clearly above chance and frequent, so it needs no hand-picked minimum beyond a small floor of 5.

```
import math, random, re
from collections import Counter
from pydoc_data.topics import topics

def tokens(text):
    return re.findall(r"[a-z]+(?:'[a-z]+)?", text.lower())

docs = [tokens(doc) for doc in topics.values()]
uni, real = Counter(), Counter()
for w in docs:
```

```
    uni.update(w)
    real.update(zip(w, w[1:]))
N = sum(uni.values())

random.seed(1)
shuffled = Counter()
for w in docs:
    w = w[:]
    random.shuffle(w)                  # same words, order destroyed
    shuffled.update(zip(w, w[1:]))

print("pair             real  shuffled")
for pair in [("of", "the"), ("in", "the"), ("floating", "point"), ("except", "clause")]:
    print(f"{' '.join(pair):15} {real[pair]:5d} {shuffled[pair]:9d}")

def t_score(pair):
    expected = uni[pair[0]] * uni[pair[1]] / N
    return (real[pair] - expected) / math.sqrt(real[pair])

top = sorted((p for p in real if real[p] >= 5), key=t_score, reverse=True)[:8]
print("t-score top:", [(" ".join(p), real[p], round(t_score(p), 1)) for p in top])
# pair             real  shuffled
# of the            532       112
# in the            447       109
# floating point     36         0
# except clause      48         3
# t-score top: [('of the', 532, 18.3), ('in the', 447, 16.0), ('in version', 205, 14.0),
```

A shuffle (one seed, so one run, not an average) makes *of the* 112 times, close to the expected 109, and *floating point* zero times against 36 in the real text. The t-score list is almost the opposite of the PMI list: it is topped by *of the*, *in the* and *in version*, because those are frequent and reliably above chance, while PMI preferred rare, tightly bound pairs. Neither is wrong; they answer different questions.

| SCORE | FAVOURS | TOP OF THE LIST HERE |
|---|---|---|
| Raw | common words next to each other | of the, in the, to the |
| PMI | tightly bound but rare pairs; needs a minimum count | floating point, finally clause (minimum 30) |

| SCORE | FAVOURS | TOP OF THE LIST HERE |
|---|---|---|
| t-score | frequent pairs clearly above chance | of the, in the, in version |

Common mistakes

Forgetting that the last word of a text has no right-hand neighbour (*n* words give *n* − 1 bigrams); letting pairs run across the boundary between two unrelated documents; ranking by PMI with no minimum count and getting a list of rare words; treating a high raw count as proof of a collocation when it may only be two common words; and reading “above chance” as “meaningful”, since *of the* beats chance without being a phrase anyone would put in a dictionary.

### EXERCISE 1

Write a short invented passage of your own (two sentences). Build its bigrams and trigrams, then print how many words, bigrams and trigrams it has, how many bigrams are distinct, the commonest bigrams, and every word that follows one chosen word.

View solution

### EXERCISE 2

On the documentation text, print the six commonest bigrams by raw count, then the six commonest that contain none of a list of small common words, then the six best by PMI among pairs seen at least 10 times. Which list looks most like a list of phrases?

View solution

### EXERCISE 3

Shuffle the words inside each documentation topic twenty times, with a different random seed each time. For four chosen pairs, report the real count, the highest count any shuffle produced, and how many shuffles matched or beat the real count.

View solution

### QUICK REFERENCE

- **Bigrams, trigrams:** `zip(words, words[1:])` and `zip(words, words[1:], words[2:])`; *n* words give *n* − 1 and *n* − 2
- **Sparsity:** longer n-grams repeat less (here 53% of bigrams and 66% of trigrams were seen once)
- **Expected count of a pair:** `count(a) * count(b) / N`
- **PMI:** `math.log2(seen / expected)`; 0 is chance, 1 is twice chance; always set a minimum count
- **t-score:** `(seen - expected) / math.sqrt(seen)`; favours frequent pairs
- **Shuffle test:** `random.shuffle(words)` inside each document, then recount
- **Caution:** a pair above chance is not necessarily a phrase; look at the list yourself

### COMING NEXT

Words by the Numbers 5 is about Markov text: predicting the next word, building a Markov chain generator, and why its output sounds almost right.

# Words by the Numbers

Chapter 5 · Markov Text

Words by the Numbers 4 counted which words sit next to each other. This chapter turns those counts around: if you know the word you just read, which word is likely to come next? Answer that question with the counts and you can also run it backwards and write text, one word at a time, by rolling dice weighted by the counts. The result is a **Markov chain** text generator. It is about twenty lines of Python, it has no idea what any word means, and its output still sounds almost right, which is the interesting part. The maths is light: counting, division and multiplying probabilities.

**WHAT WAS RUN, AND WHAT TO TRUST**

Every Python snippet in this chapter and in the three exercise solutions was executed (Python 3.14), and the output in the comments is what Python printed. The real text is the same as in Words by the Numbers 4: the reference documentation that ships inside Python (the `pydoc_data.topics` module), about 74,500 word tokens of technical English. It is not typical prose, and its counts change a little between Python versions, so expect small differences on your machine. The generated passages depend on the random seed given in the code; other seeds give other text. The tiny sentences in the first snippet and in Exercise 1 are *invented, illustrative* examples. Background facts come from a web search that returned summaries of web pages; such fetched pages are often truncated or model-summarised, so treat the wording as a paraphrase. The search confirmed Markov's use of the poem *Eugene Onegin* and the figures quoted below; it did not give Shannon's exact word-level orders or example sentences, so those are left out and the reader is pointed to his paper (check). Statements marked (check) were not confirmed against a source.

## Where the Idea Came From

Andrey Markov studied sequences in which the next item depends on the current one. In a 1913 lecture he applied the idea to text: according to the search summaries, he took the first 20,000 letters of Pushkin's verse novel *Eugene Onegin*, marked each letter as a vowel or a consonant (8,638 vowels and 11,362 consonants), and showed that the chance of a vowel depends on what came before it. An English translation of the paper appeared in 2006, per the same sources. In 1948 Claude Shannon's paper *A Mathematical Theory of Communication* used the same kind of model to produce approximations of English of increasing order; sources describe them as a series of Markov-chain models, but I could not confirm the orders or the sample sentences from

a source, so look them up in the paper itself (check). Words by the Numbers 9 comes back to Shannon.

## Predicting the Next Word

The model has one rule: the next word depends only on the word just read, not on anything earlier. For each word, keep a list of every word that has followed it. The list keeps repeats, so a word that followed three times out of four is three times as likely to be picked when you choose from the list with `random.choice`. Picking from the list is the same as picking by probability.

```
import random
from collections import Counter, defaultdict

text = "the cat sat on the mat and the cat saw the dog"
words = text.split()
followers = defaultdict(list)
for a, b in zip(words, words[1:]):
    followers[a].append(b)
print(dict(followers))
counts = Counter(followers["the"])
total = sum(counts.values())
print({w: f"{c}/{total}" for w, c in counts.items()})

# chance of the path the -> cat -> sat: P(cat after the) x P(sat after cat)
p = (counts["cat"] / total) * (Counter(followers["cat"])["sat"] / len(followers["cat"]))
print("chance of cat then sat, starting from the:", p)

random.seed(4)
word, out = "the", ["the"]
for _ in range(7):
    if word not in followers:
        break                       # 'dog' ended the text: no follower seen
    word = random.choice(followers[word])
    out.append(word)
print(" ".join(out))
# {'the': ['cat', 'mat', 'cat', 'dog'], 'cat': ['sat', 'saw'], 'sat': ['on'], 'on': ['th
# {'cat': '2/4', 'mat': '1/4', 'dog': '1/4'}
# chance of cat then sat, starting from the: 0.25
```

```
# the mat and the dog
```

*the* was followed by *cat* twice, *mat* once and *dog* once, so the chance of *cat* after *the* is 2/4. To get the chance of a whole path, multiply the steps: *cat* after *the* (2/4) times *sat* after *cat* (1/2) is 1/4. The generated line also shows the model's one structural weakness: it stopped at *dog*, which ended the sample text, so no follower had ever been seen. Real generators need a rule for that; the next snippet jumps to a random word.

## A Chain Built from Real Text

Now the same idea on the whole documentation text. The snippet builds the follower lists, counts how many words have only one kind of follower, and prints three short passages. Starting words are drawn from the full list of tokens, so common words start more often. When a word has no follower the code jumps to a random word, and the lists never run across the end of one documentation topic into the next.

```
import random, re
from collections import defaultdict
from pydoc_data.topics import topics

def tokens(text):
    return re.findall(r"[a-z]+(?:'[a-z]+)?", text.lower())

followers = defaultdict(list)
starts = []
for doc in topics.values():
    w = tokens(doc)
    starts.extend(w)
    for a, b in zip(w, w[1:]):
        followers[a].append(b)

print("distinct words with a follower:", len(followers))
one = sum(1 for f in followers.values() if len(set(f)) == 1)
print("words with only one kind of follower:", f"{one / len(followers):.0%}")

random.seed(7)
for _ in range(3):
```

```
    word = random.choice(starts)
    out = [word]
    while len(out) < 18:
        word = random.choice(followers[word]) if word in followers else random.choice(st
        out.append(word)
    print(" ".join(out))
# distinct words with a follower: 3288
# words with only one kind of follower: 38%
# should look for a function or none if stmt an extra argument for example split if only
# continue id continue or methods object rshift self args because otherwise str bytes jo
# attributes the class s keys for getitem return true the traceback attached to the cont
```

Of the 3,288 words that have a follower, 38% were only ever followed by one kind of word, so the chain has no choice there. Read the passages in pieces: *an extra argument for example* and *the traceback attached to the container* are fluent fragments, and each pair of neighbouring words really occurs in the source. But no sentence lasts: the model has no memory of how the sentence began, so it wanders into something unrelated after a few words. That is the "almost right". Every pair is plausible and the whole is not. Because the source is technical documentation, the nonsense also sounds technical (*continue id continue or methods object rshift self args*).

## Short Memory, Long Memory

The fix is to remember more. An **order-*k*** chain uses the last *k* words as its context. Order 1 is what you just built; order 2 looks at the last two words (the bigrams of Words by the Numbers 4), order 3 at the last three. Generation works the same way: look up the context, pick a follower, slide the window along by one word. The snippet builds orders 1 to 3, generates 3,000 words from each, and measures two things. The first is how many contexts have only one possible follower. The second is how many 4-word runs in the generated text can be found word for word in the source, which measures copying.

```
import random, re
from collections import defaultdict
from pydoc_data.topics import topics

def tokens(text):
```

```
    return re.findall(r"[a-z]+(?:'[a-z]+)?", text.lower())

docs = [tokens(d) for d in topics.values()]

def build(order):
    chain = defaultdict(list)
    for w in docs:
        for i in range(len(w) - order):
            chain[tuple(w[i:i + order])].append(w[i + order])
    return chain

def generate(chain, order, n, seed):
    rng = random.Random(seed)
    keys = list(chain)
    state = rng.choice(keys)
    out = list(state)
    while len(out) < n:
        if state not in chain:
            state = rng.choice(keys)           # dead end: jump to a random context
            out.extend(state)
            continue
        out.append(rng.choice(chain[state]))
        state = tuple(out[-order:])
    return out[:n]

source4 = set()
for w in docs:
    source4.update(zip(w, w[1:], w[2:], w[3:]))

for order in (1, 2, 3):
    chain = build(order)
    single = sum(1 for f in chain.values() if len(set(f)) == 1) / len(chain)
    out = generate(chain, order, 3000, seed=order)
    windows = list(zip(out, out[1:], out[2:], out[3:]))
    copied = sum(1 for g in windows if g in source4) / len(windows)
    print(f"order {order}: {len(chain):6d} contexts | one possible follower: {single:4.0%
    print("   ", " ".join(generate(chain, order, 16, seed=order + 10)))
# order 1:   3288 contexts | one possible follower:  38% | 4-word runs found in source:
#     fault after the sequence methods should be supported builds a statement this rule
# order 2:  27494 contexts | one possible follower:  76% | 4-word runs found in source:
#     more will likely be left undefined object add self other modulo object ilshift sel
# order 3:  46929 contexts | one possible follower:  91% | 4-word runs found in source:
#     runcall returns it returns the value of the literal for example the annotation lis
```

As the memory grows the sentences get steadier (the order-3 sample, *returns the value of the literal for example the annotation*, reads almost like a real documentation sentence), but the freedom disappears. At order 3, 91% of the contexts have a single follower and 100% of the 4-word runs come straight from the source. The 100% is partly built in: an order-3 context plus its follower is itself a 4-word run from the source, so the figure could only fall below 100% where the code jumped to a random context. The generator has stopped writing and started quoting. Order 2 sits in the middle, with 64% of runs found in the source.

| ORDER | CONTEXTS SEEN | ONLY ONE FOLLOWER | 4-WORD RUNS FOUND IN SOURCE |
|---|---|---|---|
| 1 | 3,288 | 38% | 4% |
| 2 | 27,494 | 76% | 64% |
| 3 | 46,929 | 91% | 100% |

This is the sparsity problem from Words by the Numbers 4 seen from a new side. With about 74,500 words of source, most three-word contexts were followed by a single kind of word, so the model can only repeat what it saw. A longer memory needs far more text. The trade is general: low order gives variety with poor sense, high order gives sense by copying.

## How Good Is the Guess?

Writing text is hard to score by eye, so test the prediction itself. Hold back every fifth documentation topic as a test set, build the counts from the rest, and for each test position guess the next word. Three guessers compete: always say the word that is commonest in the training text; say the commonest follower of the last word; and say the commonest follower of the last two words, falling back to the last word alone when that pair was never seen in training. A guess counts only if it is exactly the right word, so the scores will look low.

```
import re
from collections import Counter, defaultdict
from pydoc_data.topics import topics

def tokens(text):
    return re.findall(r"[a-z]+(?:'[a-z]+)?", text.lower())

names = sorted(topics)
```

```
train = [tokens(topics[n]) for i, n in enumerate(names) if i % 5 != 0]
test = [tokens(topics[n]) for i, n in enumerate(names) if i % 5 == 0]

uni = Counter(w for d in train for w in d)
c1, c2 = defaultdict(Counter), defaultdict(Counter)
for d in train:
    for a, b in zip(d, d[1:]):
        c1[a][b] += 1
    for a, b, c in zip(d, d[1:], d[2:]):
        c2[(a, b)][c] += 1

def predict(prev2, prev1, mode):
    if mode >= 2 and (prev2, prev1) in c2:
        return c2[(prev2, prev1)].most_common(1)[0][0]
    if mode >= 1 and prev1 in c1:
        return c1[prev1].most_common(1)[0][0]
    return uni.most_common(1)[0][0]

total = 0
hits = Counter()
seen2 = 0
for d in test:
    for a, b, c in zip(d, d[1:], d[2:]):
        total += 1
        seen2 += (a, b) in c2
        for mode in (0, 1, 2):
            hits[mode] += predict(a, b, mode) == c
print("training words:", sum(uni.values()), "| test positions:", total)
print("test positions whose 2-word context was seen in training:", f"{seen2 / total:.0%}")
for mode, label in [(0, "always the commonest word"), (1, "most common follower of last
    print(f"{label:40} {hits[mode] / total:5.1%}")
# training words: 68787 | test positions: 5719
# test positions whose 2-word context was seen in training: 67%
# always the commonest word                 6.6%
# most common follower of last word        16.5%
# last two words, fall back to last one    29.8%
```

The ranking is the lesson. Always guessing the commonest word is right 6.6% of the time. Using the last word more than doubles that, to 16.5%, and adding the second word with a fallback reaches 29.8%. Even so, the two-word context was seen in training at only 67% of test positions, so a third of the time the guesser had to fall back. A score near 30% sounds poor, but the

guesser is choosing among thousands of words in text that is not predictable by design. Guessing the next word is also the job modern language models do, with far bigger models and far more text (general background, not something I checked; check).

| GUESSER | RIGHT ON HELD-OUT TOPICS |
| --- | --- |
| Commonest word every time | 6.6% |
| Commonest follower of last word | 16.5% |
| Last two words, fall back to last one | 29.8% |

Common mistakes

Storing followers as a set, which loses the repeats and so the probabilities; forgetting the dead end at the last word of a text, where no follower exists; letting the chain run across the end of one document into the next; raising the order and mistaking copied text for a good model; judging a generator by one lucky sentence rather than by many; testing a predictor on the same text it was built from, which makes it look far better than it is; and treating a fluent sentence as a meaningful one, since the chain knows only which words sit together, never what they mean.

**EXERCISE 1**

Write a short invented passage of your own (one or two sentences). Build the follower lists for a first-order chain. Print how many words and distinct bigrams it has, the followers of three chosen words as fractions, the chance of one exact three-word path, and a short generated line.

View solution

**EXERCISE 2**

On the documentation text, for orders 1, 2 and 3, generate 2,000 words and report the number of dead-end jumps, the share of steps where the chain had a genuine choice between different followers, and the mean number of followers per context. How does the freedom fall as the order rises?

View solution

**EXERCISE 3**

Hold out every fourth documentation topic as a test set. With a first-order model, report how often the right next word is the model's single best guess, how often it is within its three best guesses, and how often the last word was never seen in training.

View solution

### QUICK REFERENCE

- **Markov chain (order 1):** the next word depends only on the current word; order *k* uses the last *k* words
- **Build:** `followers[a].append(b)` for each pair from `zip(words, words[1:])`; keep the repeats
- **Generate:** `random.choice(followers[word])`, repeat; set `random.seed(n)` to repeat a run
- **Chance of a path:** multiply the step probabilities, e.g. 2/4 × 1/2 = 1/4
- **Dead end:** a context with no follower; jump to a random context
- **Order trade-off:** low order gives variety and nonsense, high order gives fluency by copying (here, 100% of 4-word runs at order 3)
- **Test fairly:** build on some topics, score on held-out ones

### COMING NEXT

Words by the Numbers 6 is about style and authorship: function words, vocabulary richness, comparing authors, and the Federalist Papers puzzle.

# Words by the Numbers

Chapter 6 · Style and Authorship

Words by the Numbers 5 generated text from counts. This chapter runs the other way: given a text, can counts tell you who wrote it? Every writer has habits they do not notice, such as how often they reach for *upon*, *by* or *would*, and those habits show up as steady rates across hundreds of pages. The chapter uses a famous test case, the Federalist Papers, where historians disagreed for a century and a half about who wrote twelve of the essays. The maths is light: rates per thousand words, a z-score, and an average of differences. All the Python is runnable.

**WHAT WAS RUN, AND WHAT TO TRUST**

Every Python snippet in this chapter and in the three exercise solutions was executed (Python 3.14), and the output in the comments is what Python printed. The text is the plain-text *Federalist Papers* file from Project Gutenberg (eBook 1404), which the code downloads once and saves as `federalist.txt`; you need an internet connection the first time. The essays are cut at the start of the greeting and at the closing signature, so the headings and the file's footnotes are not counted. The author labels come from that file, except that the twelve disputed essays (numbers 49 to 58, 62 and 63) and the three joint ones (18 to 20) are set aside by the code. Counts depend on exactly how words are cut, so a different tokeniser gives slightly different numbers. The two-line passages in Exercise 1 are *invented, illustrative* examples. Background facts come from a web search that returned summaries of web pages; such fetched pages are often truncated or model-summarised, so treat the wording as a paraphrase. The search supported the essay counts, the disputed essay numbers, and the 1963-64 Mosteller and Wallace study with its conclusion that Madison wrote all twelve. It also showed a source conflict, noted below. Statements marked (check) were not confirmed against a source.

## A Puzzle from 1787

The Federalist Papers are 85 newspaper essays that argued for adopting the new United States Constitution. They appeared from October 1787 (the file's own dates) over the single pen name *Publius*. The writers were Alexander Hamilton, James Madison and John Jay. Lists of who wrote what were drawn up soon after, but the lists disagreed, and twelve essays (numbers 49 to 58, 62 and 63) were claimed by both Hamilton and Madison. Historians argued from the content and the political context without settling it (general background, check). In 1963-64 the statisticians Frederick Mosteller and David Wallace tried a different kind of evidence: counting words. According to the search summaries, they used Bayes' theorem to weigh the evidence from many

words, noticed that the two men used the same words at different rates, and concluded that Madison wrote all twelve. Some later work questioned this for Federalist 55, per one source; I could not confirm the details (check).

One source conflict is worth knowing before counting. Some sources say Madison wrote 14 essays, Hamilton 51, Jay 5, with 3 joint and 12 disputed (this adds up to 85). Others say Madison wrote 29, which is the same 14 plus the 12 disputed plus the 3 joint ones once the disputes are settled in his favour. Both are consistent; they count the disputed essays differently. The code below uses the first scheme, because the disputed ones are exactly what you want to keep out of the training data.

## Function Words: Habits You Cannot See

A **function word** is a small grammatical word (*the*, *of*, *by*, *to*, *upon*) as opposed to a content word that carries the topic (*treaty*, *army*). Function words suit authorship work for two reasons. They are frequent, so even one essay gives usable counts; and writers choose them without thinking, so they are hard to imitate and are less tied to the subject. The usual unit is a **rate**: uses per 1,000 words, which makes a long essay and a short one comparable. The first snippet loads the essays, prints how much text each author has, and compares rates.

```
import pathlib, re, urllib.request

URL = "https://www.gutenberg.org/cache/epub/1404/pg1404.txt"
DISPUTED = set(range(49, 59)) | {62, 63}   # the twelve disputed essays
JOINT = {18, 19, 20}                       # joint work, left out of the comparisons

def load(path="federalist.txt"):
    p = pathlib.Path(path)
    if not p.exists():                     # download once, then reuse the saved copy
        req = urllib.request.Request(URL, headers={"User-Agent": "Mozilla/5.0"})
        p.write_bytes(urllib.request.urlopen(req).read())
    raw = p.read_text(encoding="utf-8-sig").replace("\r", "")
    parts = re.split(r"^FEDERALIST No\. (\d+)\s*$", raw, flags=re.M)
    papers = {}                            # essay number -> (author, list of words)
    for k in range(1, len(parts), 2):
        n, body = int(parts[k]), parts[k + 1]
        m = re.search(r"^(HAMILTON|MADISON|JAY)$", body, re.M)
        author = m.group(1) if m else "?"
        if n in DISPUTED: author = "DISPUTED"
```

```
        if n in JOINT: author = "JOINT"
        start = body.find("To the People")
        end = body.rfind("\nPUBLIUS")
        text = body[start:end if end > start else None]
        papers[n] = (author, re.findall(r"[a-z]+(?:'[a-z]+)?", text.lower()))
    return papers

from collections import Counter

papers = load()
essays, words, pool = Counter(), Counter(), {}
for n, (author, w) in papers.items():
    essays[author] += 1
    words[author] += len(w)
    pool.setdefault(author, Counter()).update(w)

for a in ("HAMILTON", "MADISON", "JAY", "JOINT", "DISPUTED"):
    print(f"{a:9} essays {essays[a]:2d}   words {words[a]:7,d}")

print()
print(f"{'per 1000 words':14} {'Hamilton':>9} {'Madison':>9} {'Jay':>9} {'disputed':>9}"
for word in ("upon", "whilst", "while", "by", "to", "on", "would", "there"):
    rates = [1000 * pool[a][word] / words[a] for a in ("HAMILTON", "MADISON", "JAY", "DI
    print(f"{word:14}", *[f"{r:9.2f}" for r in rates])

for a in ("HAMILTON", "MADISON"):
    own = [w for author, w in papers.values() if author == a]
    print(a, "essays with no 'upon' at all:", sum(1 for w in own if "upon" not in w), "o
# HAMILTON  essays 51   words 111,764
# MADISON   essays 14   words  38,890
# JAY       essays  5   words   8,426
# JOINT     essays  3   words   5,653
# DISPUTED  essays 12   words  24,165
#
# per 1000 words  Hamilton   Madison       Jay  disputed
# upon               3.33      0.18      0.12      0.12
# whilst             0.01      0.31      0.00      0.37
# while              0.32      0.00      0.24      0.00
# by                 7.56     11.62      9.73     11.50
# to                40.72     32.42     34.77     31.74
# on                 3.36      7.59      5.22      7.82
# would              8.25      4.09      8.07      4.55
# there              3.39      0.85      1.19      1.32
# HAMILTON essays with no 'upon' at all: 0 of 51
```

```
# MADISON essays with no 'upon' at all: 11 of 14
```

Hamilton uses *upon* 3.33 times per 1,000 words and Madison 0.18, about eighteen times less often. Eleven of Madison's 14 essays do not contain *upon* at all, while all 51 of Hamilton's do. Madison prefers *whilst* where Hamilton says *while*, and Madison writes *by* and *on* more, Hamilton *to*, *would* and *there*. Now look at the last column. The twelve disputed essays use *upon* at 0.12 per 1,000 words, *whilst* at 0.37 and *by* at 11.50. Those figures sit with Madison's, not Hamilton's. Jay, with only five essays, looks different again (for example a low *the*, which would need a closer look; check).

| PER 1,000 WORDS | HAMILTON | MADISON | DISPUTED ESSAYS |
|---|---|---|---|
| upon | 3.33 | 0.18 | 0.12 |
| whilst | 0.01 | 0.31 | 0.37 |
| by | 7.56 | 11.62 | 11.50 |
| to | 40.72 | 32.42 | 31.74 |

One word is a clue, not a proof. The disputed essays are about the structure of the new government (the code does not read them, so that is background; check), and topic can move even function-word rates a little. That is why serious studies use many words at once, as the third snippet does.

## Vocabulary Richness

A second idea is **vocabulary richness**: does one author use a wider range of words? The simplest measure is the **type-token ratio**, the number of distinct words divided by the total number of words (compare the vocabulary counts in Words by the Numbers 1). It has a well-known flaw, which the next snippet measures: the longer the text, the more words repeat, so the ratio falls. To compare writers fairly, cut every essay to the same length first. The snippet also counts the share of the 1,000 words that occur only once in the sample.

```
import pathlib, re, urllib.request

URL = "https://www.gutenberg.org/cache/epub/1404/pg1404.txt"
DISPUTED = set(range(49, 59)) | {62, 63}   # the twelve disputed essays
JOINT = {18, 19, 20}                       # joint work, left out of the comparisons

def load(path="federalist.txt"):
    p = pathlib.Path(path)
    if not p.exists():                     # download once, then reuse the saved copy
        req = urllib.request.Request(URL, headers={"User-Agent": "Mozilla/5.0"})
        p.write_bytes(urllib.request.urlopen(req).read())
    raw = p.read_text(encoding="utf-8-sig").replace("\r", "")
    parts = re.split(r"^FEDERALIST No\. (\d+)\s*$", raw, flags=re.M)
    papers = {}                            # essay number -> (author, list of words)
    for k in range(1, len(parts), 2):
        n, body = int(parts[k]), parts[k + 1]
        m = re.search(r"^(HAMILTON|MADISON|JAY)$", body, re.M)
        author = m.group(1) if m else "?"
        if n in DISPUTED: author = "DISPUTED"
        if n in JOINT: author = "JOINT"
        start = body.find("To the People")
        end = body.rfind("\nPUBLIUS")
        text = body[start:end if end > start else None]
        papers[n] = (author, re.findall(r"[a-z]+(?:'[a-z]+)?", text.lower()))
    return papers

import statistics
from collections import Counter

papers = load()
rows = [(len(w), len(set(w)) / len(w)) for a, w in papers.values() if a in ("HAMILTON",
r = statistics.correlation([n for n, _ in rows], [t for _, t in rows])
print(f"{len(rows)} essays, lengths {min(n for n, _ in rows)} to {max(n for n, _ in rows
print(f"correlation between essay length and whole-essay type-token ratio: {r:.2f}")

print()
print("first 1,000 words of each essay that is at least 1,000 words long")
for a in ("HAMILTON", "MADISON", "JAY", "DISPUTED"):
    ws = [w[:1000] for author, w in papers.values() if author == a and len(w) >= 1000]
    ttr = [len(set(w)) / 1000 for w in ws]
    once = [sum(1 for c in Counter(w).values() if c == 1) / 1000 for w in ws]
    print(f"{a:9} essays {len(ws):2d}  distinct {statistics.mean(ttr):.3f} (spread {stat
# 70 essays, lengths 969 to 5725 words
```

```
# correlation between essay length and whole-essay type-token ratio: -0.76
#
# first 1,000 words of each essay that is at least 1,000 words long
# HAMILTON  essays 50  distinct 0.405 (spread 0.029)  used once 0.289
# MADISON   essays 14  distinct 0.390 (spread 0.037)  used once 0.269
# JAY       essays  5  distinct 0.407 (spread 0.028)  used once 0.282
# DISPUTED  essays 12  distinct 0.377 (spread 0.024)  used once 0.249
```

The correlation of -0.76 says that longer essays have lower ratios, so comparing whole essays would mostly compare their lengths. With every essay cut to 1,000 words the ratios are close: Hamilton 0.405, Jay 0.407 and Madison 0.390. The spread between essays by the same author (0.03 to 0.04) is about as big as the gap between the authors. So vocabulary richness does not separate these writers, and it is a weaker tool here than function-word rates. The disputed essays score 0.377, a little under Madison's average but within the spread; I would not lean on that. A measure that cannot beat its own noise is a result worth reporting, not hiding.

## Comparing Authors: A Simple Delta

To use many words at once, put each essay on a common scale. For each of the *k* commonest words, take its rate in the essay and convert it to a **z-score**: how many standard deviations it is above or below the average essay (z = (rate − mean) / sd). Do the same for an author's pooled writing to get that author's profile. The distance between an essay and a profile is the average of the absolute differences between their z-scores. This is a simple version of the measure called Burrows' Delta (the name and the idea are from literary statistics; the details here are my simplified version). The smaller the Delta, the closer the essay is to that author.

Before trusting it on the disputed essays, test it where the answer is known. The snippet below leaves out one essay at a time, builds the profiles from the others, and guesses the author of the one left out. Only then does it score the twelve disputed essays, using all the known essays and the 30 commonest words.

```
import pathlib, re, urllib.request

URL = "https://www.gutenberg.org/cache/epub/1404/pg1404.txt"
DISPUTED = set(range(49, 59)) | {62, 63}   # the twelve disputed essays
```

```
JOINT = {18, 19, 20}                       # joint work, left out of the comparisons

def load(path="federalist.txt"):
    p = pathlib.Path(path)
    if not p.exists():                     # download once, then reuse the saved copy
        req = urllib.request.Request(URL, headers={"User-Agent": "Mozilla/5.0"})
        p.write_bytes(urllib.request.urlopen(req).read())
    raw = p.read_text(encoding="utf-8-sig").replace("\r", "")
    parts = re.split(r"^FEDERALIST No\. (\d+)\s*$", raw, flags=re.M)
    papers = {}                            # essay number -> (author, list of words)
    for k in range(1, len(parts), 2):
        n, body = int(parts[k]), parts[k + 1]
        m = re.search(r"^(HAMILTON|MADISON|JAY)$", body, re.M)
        author = m.group(1) if m else "?"
        if n in DISPUTED: author = "DISPUTED"
        if n in JOINT: author = "JOINT"
        start = body.find("To the People")
        end = body.rfind("\nPUBLIUS")
        text = body[start:end if end > start else None]
        papers[n] = (author, re.findall(r"[a-z]+(?:'[a-z]+)?", text.lower()))
    return papers

import statistics
from collections import Counter

papers = load()
known = {n: a for n, (a, w) in papers.items() if a in ("HAMILTON", "MADISON")}
counts = {n: Counter(w) for n, (a, w) in papers.items()}
size = {n: len(w) for n, (a, w) in papers.items()}

def make_model(train, k):
    pooled = Counter()
    for n in train:
        pooled.update(counts[n])
    vocab = [w for w, _ in pooled.most_common(k)]          # the k commonest words
    rates = [[counts[n][v] / size[n] for v in vocab] for n in train]
    mean = [statistics.mean(col) for col in zip(*rates)]
    sd = [statistics.pstdev(col) for col in zip(*rates)]
    profiles = {}
    for author in ("HAMILTON", "MADISON"):
        own = Counter()
        for n in train:
            if known[n] == author:
                own.update(counts[n])
        total = sum(own.values())
```

```
        profiles[author] = [(own[v] / total - mean[i]) / sd[i] for i, v in enumerate(voc
    return vocab, mean, sd, profiles

def delta(n, model):
    vocab, mean, sd, profiles = model
    z = [(counts[n][v] / size[n] - mean[i]) / sd[i] for i, v in enumerate(vocab)]
    return {a: statistics.mean(abs(z[i] - p[i]) for i in range(len(vocab))) for a, p in

K = 30
right, seen = Counter(), Counter()
for n, a in known.items():                                  # leave one essay out, then
    model = make_model([m for m in known if m != n], K)
    d = delta(n, model)
    seen[a] += 1
    right[a] += min(d, key=d.get) == a
print("test on essays of known authorship:", dict(right), "right out of", dict(seen))

model = make_model(list(known), K)
print("the", K, "words used:", " ".join(model[0]))
tally = Counter()
for n in sorted(papers):
    if papers[n][0] == "DISPUTED":
        d = delta(n, model)
        best = min(d, key=d.get)
        tally[best] += 1
        print(f"essay {n}: Hamilton {d['HAMILTON']:.2f}  Madison {d['MADISON']:.2f}  -> 
print(dict(tally))
# test on essays of known authorship: {'HAMILTON': 49, 'MADISON': 12} right out of {'HAM
# the 30 words used: the of to and in a be that it is which as by this would have or for
# essay 49: Hamilton 0.96  Madison 0.87  -> MADISON
# essay 50: Hamilton 1.13  Madison 0.99  -> MADISON
# essay 51: Hamilton 1.15  Madison 1.01  -> MADISON
# essay 52: Hamilton 0.76  Madison 0.67  -> MADISON
# essay 53: Hamilton 0.85  Madison 0.66  -> MADISON
# essay 54: Hamilton 1.05  Madison 0.85  -> MADISON
# essay 55: Hamilton 0.76  Madison 0.85  -> HAMILTON
# essay 56: Hamilton 1.12  Madison 1.06  -> MADISON
# essay 57: Hamilton 1.04  Madison 0.80  -> MADISON
# essay 58: Hamilton 0.79  Madison 0.60  -> MADISON
# essay 62: Hamilton 0.81  Madison 0.64  -> MADISON
# essay 63: Hamilton 0.64  Madison 0.52  -> MADISON
# {'MADISON': 11, 'HAMILTON': 1}
```

On essays whose authors are known, the method names 49 of Hamilton's 51 and 12 of Madison's 14 correctly, 61 out of 65 (94%). That is the honest test, and it comes before looking at the disputed essays. For the twelve disputed essays, eleven are closer to Madison's profile and one, essay 55, is closer to Hamilton's, by a margin of 0.09 (0.76 against 0.85). That essay is also the one the search summary flagged as doubtful, which is interesting but does not prove anything: with 94% accuracy on known essays, about one wrong call in twelve is about what that accuracy leads you to expect, and the code cannot tell you why it was wrong. Note also what is missing: this method gives no probability, only a nearer or farther.

| WORDS USED (COMMONEST K) | KNOWN ESSAYS RIGHT (OF 65) | DISPUTED ESSAYS CLOSER TO MADISON |
|---|---|---|
| 10 | 55 | 8 of 12 |
| 30 | 61 | 11 of 12 |
| 100 | 61 | 11 of 12 |
| 200 | 61 | 12 of 12 |

The table comes from Exercise 3. With 10 words the test accuracy is lower (55 of 65) and the disputed verdicts wobble; from 30 words up, the test accuracy is steady and the verdicts mostly agree. A result that survives changing the settings is more believable than one that depends on a lucky choice. This does not reproduce Mosteller and Wallace's work, which used selected marker words and Bayesian probabilities; it only shows that a much simpler method points the same way (their method as summarised in the search; check).

Common mistakes

Training on the essays you are testing, which makes the method look perfect; leaving the disputed essays in the training data; comparing whole texts of different lengths with a length-sensitive measure such as the type-token ratio; using topic words (*department*, *confederation*) as if they were style, since they change with the subject; relying on one marker word; reading a small margin as a verdict; having very few essays from one author (Jay has five), so his profile is unreliable; and forgetting that the numbers measure similarity to a profile, not who actually sat at the desk. Authorship statistics give evidence, not certainty.

**EXERCISE 1**

Write two short invented passages by two imaginary authors and one mystery passage. Choose ten function words, compute each passage's rate per 100 words, add up the absolute differences between

the mystery rates and each author's, and say which author is closer. Then say why such short samples prove very little.

View solution

### EXERCISE 2

On the Federalist text, find the words (used at least 40 times in total by Hamilton and Madison together) whose rates differ most between the two. Use the base-2 logarithm of the ratio of rates, print the top eight each way, and decide which of them look like style and which like topic.

View solution

### EXERCISE 3

Repeat the leave-one-out test with the 10, 30, 100 and 200 commonest words. For each, report how many of the 65 known essays are guessed correctly and how many of the twelve disputed essays come out closer to Madison. Is the answer stable?

View solution

### QUICK REFERENCE

- **Function words:** small grammatical words; frequent, unconscious, less tied to topic
- **Rate:** `1000 * count / total_words`, so essays of different lengths compare
- **Type-token ratio:** `len(set(words)) / len(words)`; falls as the text grows, so cut to equal length (here, r = -0.76 with length)
- **z-score:** `(rate - mean) / sd` for each word
- **Simple Delta:** the mean of `abs(z_essay - z_author)` over the *k* commonest words; smaller is closer
- **Honest test:** leave one known essay out, rebuild the profiles, guess it (61 of 65 here)
- **Check robustness:** change *k* and see whether the verdicts move
- **Source conflict:** Madison's essay count is 14 or 29 depending on whether the disputed and joint essays are included

### COMING NEXT

Words by the Numbers 7 is about readability: sentence and word length, Flesch-style formulas, and what the scores miss.

# Words by the Numbers

Chapter 7 · Readability

Words by the Numbers 6 asked who wrote a text. This chapter asks a plainer question: how hard is it to read? Two things are easy to count, how long the sentences are and how long the words are, and for about 75 years people have turned those two counts into a single score. The maths is a weighted sum, so it fits on one line, and the Python is runnable. The more interesting half of the chapter is testing what the score does not see.

**WHAT WAS RUN, AND WHAT TO TRUST**

Every Python snippet in this chapter and in the three exercise solutions was executed (Python 3.14), and the output in the comments is what Python printed. The four books are plain-text files from Project Gutenberg (eBook numbers 11, 35, 1342 and 1404), which the code downloads once and saves next to the script; you need an internet connection the first time. Everything between the Gutenberg start and end markers is counted, so title pages, contents lists, headings and footnotes are in the totals; headings without a full stop run into the sentence after them. The scores are therefore approximate, and a different tokeniser, sentence splitter or syllable counter gives different numbers (that is one of the points of the chapter). The sentences in the first snippet, the two passages at the end of the third snippet and the 100-word example are *invented, illustrative* examples. The syllable counts in the test list are my own hand counts, and a few words can be counted two ways. The history comes from a web search that returned summaries of web pages; such fetched pages are often truncated or model-summarised, so treat the wording as a paraphrase. The search supported the two formulas, the 1948 date for Flesch's test and the 1975 Navy study, and it showed two source conflicts, noted below. Statements marked (check) were not confirmed against a source.

## A Yardstick from 1948

Rudolf Flesch published his Reading Ease test in 1948, according to the search summaries (one of them places it in his book *The Art of Readable Writing*; I could not confirm which publication first printed it, check). The idea was that a writer could measure a draft instead of guessing. The grade-level version came later: it was developed under a US Navy contract in 1975 by J. Peter Kincaid and colleagues, so that the Navy could judge whether its training manuals matched the reading skills of recruits. The sources disagree on two details. One dates the Navy study to 1976 rather than 1975. And one says Flesch himself made the grade-level conversion, while another credits the joint Flesch and Kincaid work. Other formulas exist (Gunning Fog, SMOG and Dale-

Chall are names I know of, but I did not check their details here; check). All of them are built from the same kind of ingredients.

## The Formula, Worked by Hand

Both Flesch-style scores use only two ratios: the average number of words per sentence, and the average number of syllables per word. Per the search, the Reading Ease formula is `206.835 − 1.015 × (words per sentence) − 84.6 × (syllables per word)`, and the Grade Level formula is `0.39 × (words per sentence) + 11.8 × (syllables per word) − 15.59`. Try them on an invented sample of 100 words in 5 sentences with 140 syllables. That is 20 words per sentence and 1.4 syllables per word. Reading Ease: 206.835 − 1.015 × 20 − 84.6 × 1.4 = 206.835 − 20.3 − 118.44 = 68.1. Grade Level: 0.39 × 20 + 11.8 × 1.4 − 15.59 = 7.8 + 16.52 − 15.59 = 8.7. Exercise 1 explores how the score moves when you change either ratio.

| | READING EASE | GRADE LEVEL |
|---|---|---|
| Weight on words per sentence | −1.015 | +0.39 |
| Weight on syllables per word | −84.6 | +11.8 |
| Constant | 206.835 | −15.59 |
| Higher score means | easier to read | harder to read |
| Reads as | a score, often described as 0 to 100 | a US school grade |

Notice how lopsided the weights are. One more syllable per word on average matters about 83 times more than one more word per sentence in Reading Ease (84.6 against 1.015), though the two quantities are on different scales, so the sensible comparison is a realistic nudge: 0.1 more syllables per word costs 8.5 points, one more word in every sentence only 1.0. In practice the word-length term dominates. The formulas assume that long words are hard and long sentences are hard; both are statistical tendencies in ordinary prose and not laws of reading.

## Counting Sentences, Words and Syllables

The formula needs three counts, and each is a judgement. A **sentence** is cut at `.`, `!` or `?` followed by a space. A **word** is a run of letters, with an inner apostrophe allowed (compare the tokenising in Words by the Numbers 1). A **syllable** is the hard one: English spelling does not say

how many there are. The classic shortcut is to count runs of vowel letters (*a e i o u y*) and then fix the commonest exceptions, such as a silent final *e* and the *-ed* ending in *walked*. The snippet below does that, tests the counter on 28 words I counted by hand, and scores two invented passages.

```
import re

def syllables(word):
    w = re.sub(r"[^a-z]", "", word.lower())
    if not w:
        return 0
    n = len(re.findall(r"[aeiouy]+", w))        # runs of vowels = syllables, roughly
    if w.endswith("e") and not w.endswith("le") and n > 1:
        n -= 1                                  # a silent final e, as in "make"
    if w.endswith("ed") and not w.endswith(("ted", "ded")) and n > 1:
        n -= 1                                  # "walked" is one syllable, "wanted" two
    return max(n, 1)

# Counts made by hand (my own reading of the words); the code is tested against them
HAND = {"the": 1, "make": 1, "table": 2, "reading": 2, "sentence": 2, "people": 2,
        "beautiful": 3, "readability": 5, "syllable": 3, "area": 3, "idea": 3,
        "quiet": 2, "science": 2, "poem": 2, "created": 3, "walked": 1,
        "little": 2, "different": 3, "something": 2, "wanted": 2,
        "chocolate": 3, "queue": 1, "jumped": 1, "ocean": 2, "really": 2,
        "average": 3, "bakery": 3, "today": 2}

wrong = {w: (syllables(w), n) for w, n in HAND.items() if syllables(w) != n}
print(f"{len(HAND) - len(wrong)} of {len(HAND)} words right")
for w, (got, want) in wrong.items():
    print(f"  {w}: counted {got}, hand count {want}")

def stats(text):
    sentences = [s for s in re.split(r"(?<=[.!?])\s+", text.strip()) if s]
    words = re.findall(r"[A-Za-z]+(?:'[A-Za-z]+)?", text)
    syl = sum(syllables(w) for w in words)
    return len(sentences), len(words), syl

def flesch(text):
    s, w, y = stats(text)
    ease = 206.835 - 1.015 * (w / s) - 84.6 * (y / w)
    grade = 0.39 * (w / s) + 11.8 * (y / w) - 15.59
    return ease, grade
```

```
samples = {                                    # invented, illustrative
    "plain": "The cat sat on the mat. It was a warm day. The dog ran to the gate.",
    "dense": "Notwithstanding considerable organisational reservations, the committee un
}
for name, text in samples.items():
    s, w, y = stats(text)
    ease, grade = flesch(text)
    print(f"{name:6} sentences {s}  words {w}  syllables {y}  ease {ease:6.1f}  grade {g
# 21 of 28 words right
#   area: counted 2, hand count 3
#   idea: counted 2, hand count 3
#   quiet: counted 1, hand count 2
#   science: counted 1, hand count 2
#   poem: counted 1, hand count 2
#   created: counted 2, hand count 3
#   something: counted 3, hand count 2
# plain  sentences 3  words 17  syllables 17  ease  116.5  grade  -1.6
# dense  sentences 1  words 13  syllables 52  ease -144.8  grade  36.7
```

The counter gets 21 of 28 test words right (75%), which is a sensible thing to report before using it. Its errors have a pattern: vowels that sit side by side but belong to different syllables (*area*, *idea*, *quiet*, *science*, *poem*, *created*) are merged into one, and a silent *e* in the middle of a word (*something*) is counted. Errors in both directions partly cancel over a long text, but they do not cancel exactly. The two invented passages show the scale. The plain one (17 words, 17 syllables) scores 116.5 for ease, above the 100 that people often quote as the top, and a grade of −1.6. The dense one has a single 13-word sentence of 52 counted syllables and scores −144.8, a grade of 36.7. The formulas are weighted sums, so nothing stops them leaving the range; scores far outside it mean the text is outside what the formula was designed for, and they should not be read as real grades.

## Four Books on the Yardstick

Now real text. The next snippet downloads four books and prints, for each, the two raw ratios and the two scores. Seeing the raw ratios next to the scores shows what is driving each result.

```
import pathlib, re, urllib.request
```

```
BOOKS = {"Alice in Wonderland": 11, "The Time Machine": 35,
         "Pride and Prejudice": 1342, "The Federalist Papers": 1404}

def load(number):
    path = pathlib.Path(f"pg{number}.txt")
    if not path.exists():                       # download once, then reuse the saved co
        url = f"https://www.gutenberg.org/cache/epub/{number}/pg{number}.txt"
        req = urllib.request.Request(url, headers={"User-Agent": "Mozilla/5.0"})
        path.write_bytes(urllib.request.urlopen(req).read())
    raw = path.read_text(encoding="utf-8-sig").replace("\r", "")
    start = raw.index("*** START OF")
    start = raw.index("\n", start) + 1
    end = raw.index("*** END OF")
    return raw[start:end]

def syllables(word):
    w = re.sub(r"[^a-z]", "", word.lower())
    if not w:
        return 0
    n = len(re.findall(r"[aeiouy]+", w))
    if w.endswith("e") and not w.endswith("le") and n > 1:
        n -= 1
    if w.endswith("ed") and not w.endswith(("ted", "ded")) and n > 1:
        n -= 1
    return max(n, 1)

def stats(text):
    for fancy, plain in (("’", "'"), ("“", '"'), ("”", '"')):
        text = text.replace(fancy, plain)       # curly quotes to plain ones
    text = re.sub(r"\s+", " ", text)            # one long line, so line breaks do not m
    sentences = [s for s in re.split(r"(?<=[.!?])[\"')\]]*\s+", text.strip()) if s]
    words = re.findall(r"[A-Za-z]+(?:'[A-Za-z]+)?", text)
    return len(sentences), len(words), sum(syllables(w) for w in words)

def flesch(text):
    s, w, y = stats(text)
    return 206.835 - 1.015 * (w / s) - 84.6 * (y / w), 0.39 * (w / s) + 11.8 * (y / w) -

print(f"{'book':22} {'words':>7} {'w/sent':>7} {'syl/word':>9} {'ease':>6} {'grade':>6}"
for title, number in BOOKS.items():
    text = load(number)
    s, w, y = stats(text)
    ease, grade = flesch(text)
    print(f"{title:22} {w:7,d} {w / s:7.1f} {y / w:9.3f} {ease:6.1f} {grade:6.1f}")
# book                   words  w/sent  syl/word   ease  grade
```

```
# Alice in Wonderland     26,776    16.5     1.283   81.6    6.0
# The Time Machine        32,772    16.8     1.397   71.5    7.5
# Pride and Prejudice    127,864    17.6     1.448   66.5    8.3
# The Federalist Papers  192,302    31.9     1.638   35.9   16.2
```

| BOOK | WORDS PER SENTENCE | SYLLABLES PER WORD | EASE | GRADE |
|---|---|---|---|---|
| Alice in Wonderland | 16.5 | 1.283 | 81.6 | 6.0 |
| The Time Machine | 16.8 | 1.397 | 71.5 | 7.5 |
| Pride and Prejudice | 17.6 | 1.448 | 66.5 | 8.3 |
| The Federalist Papers | 31.9 | 1.638 | 35.9 | 16.2 |

The first three books have almost the same sentence length (16.5 to 17.6 words), so their order is set by word length: 1.283, 1.397 and 1.448 syllables per word. The Federalist essays (the newspaper essays of 1787 onward from Words by the Numbers 6) are in a different league on both counts, with sentences nearly twice as long and the longest words, giving a grade of 16.2, roughly university level on this scale. That ordering matches what a reader would expect, which is the real justification for the method: it ranks well, even if the individual numbers are rough. Treat a grade of 8.3 as "about like Pride and Prejudice and unlike the Federalist", not as a measured fact about a school year.

## What the Scores Miss

A readability score is a summary of two averages, so anything that does not change those averages is invisible to it. The last snippet runs four experiments on Pride and Prejudice: shuffling the words inside each sentence, changing how sentences are cut, looking at the spread inside one book, and scoring two invented passages that the formula misjudges.

```
import pathlib, re, urllib.request

BOOKS = {"Alice in Wonderland": 11, "The Time Machine": 35,
         "Pride and Prejudice": 1342, "The Federalist Papers": 1404}

def load(number):
```

```
    path = pathlib.Path(f"pg{number}.txt")
    if not path.exists():                        # download once, then reuse the saved co
        url = f"https://www.gutenberg.org/cache/epub/{number}/pg{number}.txt"
        req = urllib.request.Request(url, headers={"User-Agent": "Mozilla/5.0"})
        path.write_bytes(urllib.request.urlopen(req).read())
    raw = path.read_text(encoding="utf-8-sig").replace("\r", "")
    start = raw.index("*** START OF")
    start = raw.index("\n", start) + 1
    end = raw.index("*** END OF")
    return raw[start:end]

def syllables(word):
    w = re.sub(r"[^a-z]", "", word.lower())
    if not w:
        return 0
    n = len(re.findall(r"[aeiouy]+", w))
    if w.endswith("e") and not w.endswith("le") and n > 1:
        n -= 1
    if w.endswith("ed") and not w.endswith(("ted", "ded")) and n > 1:
        n -= 1
    return max(n, 1)

def stats(text):
    for fancy, plain in (("’", "'"), ("“", '"'), ("”", '"')):
        text = text.replace(fancy, plain)        # curly quotes to plain ones
    text = re.sub(r"\s+", " ", text)             # one long line, so line breaks do not m
    sentences = [s for s in re.split(r"(?<=[.!?])[\"')\]]*\s+", text.strip()) if s]
    words = re.findall(r"[A-Za-z]+(?:'[A-Za-z]+)?", text)
    return len(sentences), len(words), sum(syllables(w) for w in words)

def flesch(text):
    s, w, y = stats(text)
    return 206.835 - 1.015 * (w / s) - 84.6 * (y / w), 0.39 * (w / s) + 11.8 * (y / w) -

import random, statistics

text = load(1342)                                # Pride and Prejudice
ease, grade = flesch(text)
print(f"original text: ease {ease:.1f}, grade {grade:.1f}")

# 1. Shuffle the words inside every sentence: the score cannot change
random.seed(1)
def scramble(sentence):
    ws = sentence.split()
    random.shuffle(ws)
```

```
    return " ".join(ws)
plain = re.sub(r"\s+", " ", text)
sents = re.split(r"(?<=[.!?])\s+", plain.strip())
mixed = " ".join(scramble(s) for s in sents)
ease2, grade2 = flesch(mixed)
print(f"words shuffled inside each sentence: ease {ease2:.1f}, grade {grade2:.1f}")
print("example:", scramble(sents[1000]))

# 2. The sentence splitter is a hidden choice: "Mr." and "Mrs." look like full stops
abbr = len(re.findall(r"\b(?:Mr|Mrs|Dr|Ms|St)\.", text))
guarded = re.sub(r"\b(Mr|Mrs|Dr|Ms|St)\.", r"\1", text)
ease3, grade3 = flesch(guarded)
print(f"abbreviation full stops found: {abbr}")
print(f"with them removed: ease {ease3:.1f}, grade {grade3:.1f}")

# 3. How much does the score move from place to place inside one book?
words = text.split()
chunks = [" ".join(words[i:i + 1000]) for i in range(0, len(words) - 1000, 1000)]
grades = [flesch(c)[1] for c in chunks]
print(f"{len(chunks)} chunks of 1,000 words: grade from {min(grades):.1f} to {max(grades
      f"mean {statistics.mean(grades):.1f}, sd {statistics.stdev(grades):.1f}")

# 4. Two invented passages with a similar score but very different difficulty
easy_words_hard_idea = "If the set is not the whole, the rest is not in it, so it can no
long_words_easy_idea = "Wonderful holidays, delicious chocolate, remarkable fireworks, b
for label, t in (("short words, hard to follow", easy_words_hard_idea), ("long words, ea
    e, g = flesch(t)
    print(f"{label:28} ease {e:6.1f}  grade {g:5.1f}")
# original text: ease 66.5, grade 8.3
# words shuffled inside each sentence: ease 66.5, grade 8.3
# example: herself at week, receive them calculated Netherfield on the could before. wit
# abbreviation full stops found: 1167
# with them removed: ease 63.1, grade 9.7
# 127 chunks of 1,000 words: grade from 4.5 to 15.7, mean 8.6, sd 2.1
# short words, hard to follow  ease   98.5  grade   4.2
# long words, easy to follow   ease  -65.5  grade  24.6
```

| PRIDE AND PREJUDICE, SAME TEXT | EASE | GRADE |
|---|---|---|
| As downloaded | 66.5 | 8.3 |
| Words shuffled inside each sentence | 66.5 | 8.3 |

| PRIDE AND PREJUDICE, SAME TEXT | EASE | GRADE |
|---|---|---|
| Full stops after Mr, Mrs and similar removed | 63.1 | 9.7 |
| 1,000-word pieces (127 of them) | grade from 4.5 to 15.7, mean 8.6, sd 2.1 | |

Four lessons. First, the shuffled text keeps exactly the same score, 66.5 and 8.3, though it has become nonsense: the formula never looks at word order, grammar or meaning. Second, the sentence splitter is a hidden choice. The book has 1,167 abbreviation full stops (*Mr.*, *Mrs.* and so on), and each one is counted as a sentence end by the simple splitter. Removing them moves the grade from 8.3 to 9.7, a difference as big as the gap between Alice in Wonderland and The Time Machine in the table above. Third, one book is not one number: 1,000-word pieces of the same novel range from grade 4.5 to 15.7, a standard deviation of 2.1 grades, so a single average hides a lot (my guess is that dialogue pages score low and long reflective paragraphs high, but I did not check this). Fourth, the two invented passages disagree with how a reader would judge them. A sentence in short words that is hard to follow (*If the set is not the whole, the rest is not in it, so it can not have it.*) scores 98.5 for ease, a grade of 4.2, because every word is short; a list of long but familiar words (*Wonderful holidays, delicious chocolate, remarkable fireworks, beautiful family gatherings.*) scores −65.5, a grade of 24.6. Which passage is harder is my judgement and not a measurement, but I expect most readers to agree that the formula has it backwards. Familiar long words, an unfamiliar topic, layout, the reader's background and the structure of an argument all matter, and none of them is in the formula.

Common mistakes

Treating a score as a measurement of understanding rather than of surface length; comparing scores from different tools without noting that their sentence splitters and syllable counters differ; scoring a text with headings, lists, tables or code in it as if it were prose; scoring very short texts, where one sentence moves the average; reading a grade of 24 or an ease of −145 as a real level; rewriting a text to hit a number (chopping sentences into fragments and swapping words for shorter ones raises the score and can make the writing worse); trusting the syllable counter without testing it on words you counted yourself; and reporting one figure for a long document when its parts range widely.

**EXERCISE 1**

Write a function for the Reading Ease formula that takes the number of words, sentences and syllables. Check it on the 100-word example (5 sentences, 140 syllables), then print a small table of scores for 10, 20 and 30 words per sentence with 1.2, 1.5 and 1.8 syllables per word. Solve the formula for the

syllables per word needed to score 60 at 15 words per sentence, and say which change moves the score more.

View solution

### EXERCISE 2

Compare two syllable counters on the four books: the careful one from this chapter and a naive one that only counts runs of vowel letters. Print the grade level from each, and the difference. Does the choice of counter change the order of the books? Does it change the size of the gaps between them?

View solution

### EXERCISE 3

Cut each book into 1,000-word pieces and score every piece. For each book, print the number of pieces, the mean and standard deviation of the grade, the lowest and highest, and the share of pieces above grade 10. Is the spread inside a book smaller or larger than the gap between the books?

View solution

### QUICK REFERENCE

- **Reading Ease:** `206.835 - 1.015 * (words / sentences) - 84.6 * (syllables / words)`; higher is easier
- **Grade Level:** `0.39 * (words / sentences) + 11.8 * (syllables / words) - 15.59`; higher is harder
- **Sentence count:** split on `.` `!` `?` plus a space; abbreviations like Mr. inflate it (grade 8.3 became 9.7 here)
- **Syllables:** count runs of vowel letters, then adjust for silent *e* and *-ed*; test against words you counted by hand (21 of 28 right here)
- **Word length dominates:** +0.1 syllables per word costs 8.5 ease points; +1 word per sentence costs 1.0
- **Blind to:** word order, grammar, meaning, topic, layout, reader background (shuffled text keeps its score)
- **Not bounded:** scores like 116.5 or −144.8 mean the text is outside the formula's design
- **Spread:** 1,000-word pieces of one novel ranged from grade 4.5 to 15.7

- **Source conflicts:** the Navy study is dated 1975 or 1976; the grade conversion is credited to Flesch or to the Flesch-Kincaid work

**COMING NEXT**

Words by the Numbers 8 is about what makes a word important: document frequency, TF-IDF, keyword extraction, and a tiny search ranker.

# Words by the Numbers

Chapter 8 · What Makes a Word Important

Words by the Numbers 7 scored how hard a text is to read. This chapter asks which words in a text matter. The most frequent words are almost never the answer: *the* tops nearly every count, as Words by the Numbers 2 showed, and tells you nothing about a text. What you want is a word that is frequent *here* and rare *elsewhere*. Putting those two ideas into one number gives TF-IDF, the weighting behind keyword extraction and a good deal of classic search. The maths is one logarithm and one multiplication, and by the end you will have a working search ranker in a few dozen lines.

**WHAT WAS RUN, AND WHAT TO TRUST**

Every Python snippet in this chapter and in the three exercise solutions was executed (Python 3.14.0), and the output in the comments is what Python printed. The four books are plain-text files from Project Gutenberg (eBook numbers 11, 35, 1342 and 1404), which the code downloads once and saves next to the script; you need an internet connection the first time. Everything between the Gutenberg start and end markers is used, so title pages, contents lists and illustration markers are in the data (one of them shows up as a "keyword" in Exercise 2). The four mini-documents in the first snippet are *invented, illustrative* examples, and so are the search queries, which I chose, together with the book I expected each to come from. The tokeniser is a simple lower-casing regular expression, so other tokenisers would give different counts. The history comes from web searches that returned summaries of web pages; such fetched pages are often truncated or model-summarised, so treat the wording as a paraphrase. The searches supported the title, journal and date of the 1972 paper that introduced inverse document frequency; they supported Luhn's role only through one explainer page, and they showed one source conflict, noted below. Statements marked (check) were not confirmed against a source.

## Frequent Here, Rare Elsewhere

The idea of weighting a word by how few documents contain it comes from Karen Spärck Jones. Her paper "A statistical interpretation of term specificity and its application in retrieval" appeared in the *Journal of Documentation* in 1972 (volume 28, pages 11 to 21, per the search). It counted the documents in a collection that contain a term and gave rare terms more weight; the name *inverse document frequency* came later. One explainer page credits the other half of the recipe, counting how often a term occurs in a document, to Hans Peter Luhn of IBM in the 1950s

(the date 1957 appeared in my query, not in a page I could read, and only that one page mentioned Luhn, so check it). Search engines went on to refine the weighting. A family of formulas called BM25, from the Okapi system at City University London, is a descendant; the sources conflict on the date, giving 1994 in some places and 1995 in others, so I leave it at "mid 1990s". I did not check how modern search engines are built beyond that.

## The Maths, Worked by Hand

Take a collection of **N** documents. For a word in one document, three numbers matter.

- **tf**, the term frequency: how many times the word occurs in the document, divided by the document's length in words (dividing makes long and short documents comparable).
- **df**, the document frequency: how many documents contain the word at least once.
- **idf**, the inverse document frequency: `ln(N / df)`, the natural logarithm of N divided by df. A word in every document has `ln(1) = 0`; a word in one of four documents has `ln(4) = 1.386`.

The score is `tf × idf`. The logarithm is there so that rarity helps without taking over: a word in 1 document out of 1,000 is not 1,000 times as informative as a word in every document, it is worth `ln(1000) = 6.9`. The base of the logarithm only rescales every score by the same factor, so the ranking does not depend on it. The first snippet applies this to four invented mini-documents about cats, dogs and a bank.

```
import math
from collections import Counter

docs = {                                  # invented, illustrative mini-documents
    "A": "the cat sat on the mat and the cat purred",
    "B": "the dog sat on the log and the dog barked",
    "C": "the bank raised the interest rate on the loan",
    "D": "the cat chased the dog around the bank",
}
words = {name: text.split() for name, text in docs.items()}
N = len(docs)
df = Counter(w for ws in words.values() for w in set(ws))     # in how many documents do

def idf(word):
    return math.log(N / df[word])                             # natural log
```

```
print("document frequency and idf:")
for w in ("the", "sat", "cat", "dog", "bank", "purred"):
    print(f"  {w:7} df {df[w]}  idf {idf(w):.3f}")

print("top words by tf-idf:")
for name, ws in words.items():
    counts = Counter(ws)
    score = {w: (c / len(ws)) * idf(w) for w, c in counts.items()}
    top = sorted(score.items(), key=lambda kv: -kv[1])[:3]
    print(f"  {name}: " + ", ".join(f"{w} {s:.3f}" for w, s in top))
# document frequency and idf:
#   the     df 4  idf 0.000
#   sat     df 2  idf 0.693
#   cat     df 2  idf 0.693
#   dog     df 2  idf 0.693
#   bank    df 2  idf 0.693
#   purred  df 1  idf 1.386
# top words by tf-idf:
#   A: cat 0.139, mat 0.139, purred 0.139
#   B: dog 0.139, log 0.139, barked 0.139
#   C: raised 0.154, interest 0.154, rate 0.154
#   D: chased 0.173, around 0.173, cat 0.087
```

Look at document A, "the cat sat on the mat and the cat purred", which has 10 words. The table below does the sums for five of its words. The word *the* occurs 3 times, the most of any word, but it is in all four documents, so its idf is zero and so is its score. *cat* occurs twice in A and in one other document (df = 2); *mat* and *purred* occur once each and nowhere else (df = 1). The ranking comes out as a tie: a word that is twice as frequent in the document but twice as widespread (idf 0.693 against 1.386) scores exactly the same as a word that occurs once and nowhere else. That is the trade the formula makes, and the rest of the chapter is about whether it is a good one.

| WORD IN A | COUNT | TF (OF 10) | DF (OF 4) | IDF | TF × IDF |
|---|---|---|---|---|---|
| the | 3 | 0.300 | 4 | 0.000 | 0.000 |
| on | 1 | 0.100 | 3 | 0.288 | 0.029 |
| sat | 1 | 0.100 | 2 | 0.693 | 0.069 |

| WORD IN A | COUNT | TF (OF 10) | DF (OF 4) | IDF | TF × IDF |
|---|---|---|---|---|---|
| cat | 2 | 0.200 | 2 | 0.693 | 0.139 |
| mat | 1 | 0.100 | 1 | 1.386 | 0.139 |

| WEIGHT | REWARDS | FAILS ON |
|---|---|---|
| tf alone | words the author repeats | puts *the* and *of* at the top |
| idf alone | words few documents use | rewards a one-off typo as much as a key term |
| tf × idf | words repeated here and rare elsewhere | depends on what the other documents are |

## Keywords from Whole Books

Now real text. The next snippet treats each of the four books (the same ones as Words by the Numbers 7) as one document, so N = 4, and prints for each book its most frequent words next to its top words by TF-IDF. With only four documents, idf can take just four values: 0 for a word in all four books, 0.288 for three, 0.693 for two and 1.386 for one.

```
import math, pathlib, re, urllib.request
from collections import Counter

BOOKS = {"Alice in Wonderland": 11, "The Time Machine": 35,
         "Pride and Prejudice": 1342, "The Federalist Papers": 1404}

def load(number):
    path = pathlib.Path(f"pg{number}.txt")
    if not path.exists():                          # download once, then reuse the saved co
        url = f"https://www.gutenberg.org/cache/epub/{number}/pg{number}.txt"
        req = urllib.request.Request(url, headers={"User-Agent": "Mozilla/5.0"})
        path.write_bytes(urllib.request.urlopen(req).read())
    raw = path.read_text(encoding="utf-8-sig").replace("\r", "")
    start = raw.index("\n", raw.index("*** START OF")) + 1
    return raw[start:raw.index("*** END OF")]

def tokens(text):
    text = text.replace("\u2019", "'")
    return re.findall(r"[a-z]+(?:'[a-z]+)?", text.lower())
```

```
counts = {title: Counter(tokens(load(n))) for title, n in BOOKS.items()}
N = len(counts)
df = Counter(w for c in counts.values() for w in c)

def tfidf(title):
    c = counts[title]
    total = sum(c.values())
    return {w: (k / total) * math.log(N / df[w]) for w, k in c.items()}

for title, c in counts.items():
    raw = ", ".join(w for w, _ in c.most_common(6))
    top = sorted(tfidf(title).items(), key=lambda kv: -kv[1])[:6]
    print(f"{title}")
    print(f"  most frequent: {raw}")
    print(f"  tf-idf top:    " + ", ".join(w for w, _ in top))

print("words in all four books:", sum(1 for w in df if df[w] == N), "of", len(df))
print("words in only one book: ", sum(1 for w in df if df[w] == 1))
# Alice in Wonderland
#   most frequent: the, and, to, a, she, it
#   tf-idf top:    alice, mock, turtle, hatter, gryphon, rabbit
# The Time Machine
#   most frequent: the, i, and, of, a, to
#   tf-idf top:    traveller, morlocks, weena, machine, sphinx, medical
# Pride and Prejudice
#   most frequent: the, to, of, and, her, i
#   tf-idf top:    elizabeth, darcy, bennet, jane, bingley, mrs
# The Federalist Papers
#   most frequent: the, of, to, and, in, a
#   tf-idf top:    states, government, national, federal, executive, members
# words in all four books: 898 of 14115
# words in only one book:  8709
```

| BOOK | MOST FREQUENT WORDS | TOP BY TF-IDF |
|---|---|---|
| Alice in Wonderland | the, and, to, a, she, it | alice, mock, turtle, hatter, gryphon, rabbit |
| The Time Machine | the, i, and, of, a, to | traveller, morlocks, weena, machine, sphinx, medical |
| Pride and Prejudice | the, to, of, and, her, i | elizabeth, darcy, bennet, jane, bingley, mrs |

| BOOK | MOST FREQUENT WORDS | TOP BY TF-IDF |
|---|---|---|
| The Federalist Papers | the, of, to, and, in, a | states, government, national, federal, executive, members |

The difference is dramatic. Raw frequency gives the same kind of list for every book. TF-IDF gives you, at a glance, names and places in the three novels (a Mock Turtle, a Gryphon, the Morlocks, the Bennets) and the vocabulary of government in the essays. Only 898 of the 14,115 distinct words occur in all four books, so they all score zero, and 8,709 occur in just one. The method is also blunt here: with so few documents, *mrs* ranks as a keyword of Pride and Prejudice simply because the other books use it less, and the Federalist list is generic (*states*, *government*) because the other three books say little about the subject. Keywords depend on the collection you compare against, a point Exercise 2 pushes further.

## A Tiny Search Engine

Searching is keyword extraction run backwards. Instead of asking which words characterise a document, you give the words and ask which documents they characterise. The standard structure is an **inverted index**: a dictionary from each word to the documents containing it and the count in each, so a query only touches the documents that contain its words. To make the collection big enough to be interesting, the next snippet cuts every book into 500-word pieces and treats each piece as a document, 755 in all. A piece's score for a query is the sum, over the query words, of `(1 + ln(count)) × idf`. The `1 + ln(count)` is a common softening (check how widely it is used): the tenth occurrence of a word should not count ten times as much as the first.

```
import math, pathlib, re, urllib.request
from collections import Counter, defaultdict

BOOKS = {"Alice in Wonderland": 11, "The Time Machine": 35,
         "Pride and Prejudice": 1342, "The Federalist Papers": 1404}

def load(number):
    path = pathlib.Path(f"pg{number}.txt")
    if not path.exists():                        # download once, then reuse the saved co
        url = f"https://www.gutenberg.org/cache/epub/{number}/pg{number}.txt"
```

```
        req = urllib.request.Request(url, headers={"User-Agent": "Mozilla/5.0"})
        path.write_bytes(urllib.request.urlopen(req).read())
    raw = path.read_text(encoding="utf-8-sig").replace("\r", "")
    start = raw.index("\n", raw.index("*** START OF")) + 1
    text = raw[start:raw.index("*** END OF")]
    return text.translate({0x2018: 39, 0x2019: 39, 0x201c: 34, 0x201d: 34})   # curly to

def tokens(text):
    text = text.replace("\u2019", "'")
    return re.findall(r"[a-z]+(?:'[a-z]+)?", text.lower())

# Every 500-word piece of every book is one "document"
SIZE = 500
docs = []                                       # (book title, piece number, words)
for title, number in BOOKS.items():
    words = load(number).split()
    for i in range(0, len(words) - SIZE + 1, SIZE):
        docs.append((title, i // SIZE, words[i:i + SIZE]))

index = defaultdict(dict)                       # word -> {doc id: count in that doc}
for d, (_, _, ws) in enumerate(docs):
    for w, k in Counter(tokens(" ".join(ws))).items():
        index[w][d] = k
N = len(docs)

def search(query, top=3, use_idf=True):
    scores = defaultdict(float)
    for w in tokens(query):
        idf = math.log(N / len(index[w])) if (use_idf and w in index) else 1.0
        for d, k in index.get(w, {}).items():
            scores[d] += (1 + math.log(k)) * idf
    return sorted(scores.items(), key=lambda kv: -kv[1])[:top]

print(f"{N} documents, {len(index):,} distinct words")
EXPECT = {"white rabbit": "Alice in Wonderland", "time traveller": "The Time Machine",
          "darcy bingley": "Pride and Prejudice", "union states constitution": "The Fede
for query, book in EXPECT.items():
    hits = search(query, top=10)
    right = sum(docs[d][0] == book for d, _ in hits)
    print(f"\nquery: {query!r}  ({right} of top 10 from {book})")
    for d, s in hits[:3]:
        title, piece, ws = docs[d]
        found = ", ".join(f"{w} x{index[w].get(d, 0)}" for w in tokens(query))
        print(f"  {s:6.2f}  {title} #{piece}  [{found}]")
```

```
print("\nquery 'the of and to rabbit': how many of the top 10 come from Alice in Wonderl
for use_idf in (True, False):
    hits = search("the of and to rabbit", top=10, use_idf=use_idf)
    right = sum(docs[d][0] == "Alice in Wonderland" for d, _ in hits)
    print(f"  {'with idf' if use_idf else 'no idf  '}: {right} of 10")

hits = search("the of", top=len(docs))
print("query 'the of' with idf: documents with a score above zero:", sum(s > 1e-9 for _,
# 755 documents, 14,082 distinct words
#
# query: 'white rabbit'  (10 of top 10 from Alice in Wonderland)
#    17.88  Alice in Wonderland #50  [white x6, rabbit x6]
#    15.41  Alice in Wonderland #12  [white x3, rabbit x5]
#    14.87  Alice in Wonderland #0  [white x1, rabbit x9]
#
# query: 'time traveller'  (10 of top 10 from The Time Machine)
#    13.92  The Time Machine #4  [time x11, traveller x7]
#    13.79  The Time Machine #9  [time x9, traveller x7]
#    12.74  The Time Machine #2  [time x14, traveller x5]
#
# query: 'darcy bingley'  (10 of top 10 from Pride and Prejudice)
#    10.31  Pride and Prejudice #18  [darcy x6, bingley x7]
#     9.72  Pride and Prejudice #39  [darcy x5, bingley x6]
#     9.66  Pride and Prejudice #14  [darcy x4, bingley x7]
#
# query: 'union states constitution'  (10 of top 10 from The Federalist Papers)
#     9.46  The Federalist Papers #196  [union x6, states x6, constitution x4]
#     9.10  The Federalist Papers #165  [union x4, states x10, constitution x3]
#     8.59  The Federalist Papers #346  [union x1, states x14, constitution x7]
#
# query 'the of and to rabbit': how many of the top 10 come from Alice in Wonderland?
#   with idf: 10 of 10
#   no idf  : 4 of 10
# query 'the of' with idf: documents with a score above zero: 0
```

| QUERY (MINE) | EXPECTED BOOK (MINE) | TOP 10 FROM THAT BOOK |
|---|---|---|
| white rabbit | Alice in Wonderland | 10 |
| time traveller | The Time Machine | 10 |
| darcy bingley | Pride and Prejudice | 10 |

| QUERY (MINE) | EXPECTED BOOK (MINE) | TOP 10 FROM THAT BOOK |
|---|---|---|
| union states constitution | The Federalist Papers | 10 |
| the of and to rabbit, with idf | Alice in Wonderland | 10 |
| the of and to rabbit, without idf | Alice in Wonderland | 4 |

Four things to take from the output. First, the ranker works on short queries, and the counts show why: the top piece for *white rabbit* has six of each word. Second, idf is what rescues the query that is padded with function words. Without it, *the*, *of*, *and* and *to* add the same weight as *rabbit* in every piece, and only 4 of the top 10 come from the right book; with it, all 10 do, because those four words appear in so many pieces that they weigh almost nothing. The last line of output is the extreme case: for the query "the of" every piece scores exactly zero, because both words are in all 755 pieces (`ln(755 / 755) = 0`). Third, the test is easy and I rigged it a little: the four books differ sharply in vocabulary, the queries are mine, and "right book" is a stand-in for "relevant", which is not the same thing. Fourth, the snippet scores every piece of a book on its own words, with no notion of word order, phrases or meaning, so a query like "white rabbit" would also reward a piece with six *whites* about something else (that was not tested here).

Common mistakes

Computing idf from a collection too small or too odd to mean anything (four documents gives only four idf values); forgetting that keywords depend on the other documents, so the same text gets different keywords in a different collection; taking `ln(N / df)` of a word that is not in the index (a division by zero, or a quiet skip as the snippet does); comparing scores across queries as if they were probabilities (they are only for ranking within one query); leaving markup, headers and page numbers in the text, so that *illustration* becomes a keyword; ignoring word forms (*rabbit* and *rabbits* are separate words here); not normalising by document length when documents differ in size; and judging a ranker only on queries you designed to succeed.

**EXERCISE 1**

Using the four invented mini-documents, compute idf for df = 1 to 4 under two formulas: the plain `ln(N / df)` and a smoothed `ln(1 + N / df)`. Print the top three words per document under each. Which variant lets *the* back into the top three, and is that a good thing?

View solution

### EXERCISE 2

Use the 755 pieces as the collection and extract keywords for each book by adding up the TF-IDF of each word over the book's pieces. Print the top eight, first with every word allowed and then ignoring any word that occurs in more than 20% of the pieces. Which words appear in the first list and not in the second, and why did the first list differ from the one built on four documents?

View solution

### EXERCISE 3

Evaluate the search ranker on eight natural-language queries (two per book, each padded with words like *the* and *and*). For three scoring schemes (raw count, softened count, softened count times idf), print how many of the top 10 pieces come from the book you expected, with a total at the bottom. Which scheme degrades most when the query is full of common words?

View solution

### QUICK REFERENCE

- **tf:** count of the word in the document divided by the document's length
- **df:** number of documents containing the word at least once
- **idf:** `math.log(N / df)`; zero for a word in every document, `ln(4) = 1.386` for one in one of four
- **TF-IDF:** `tf * idf`; high for words frequent here and rare elsewhere
- **Collection matters:** keywords change when the other documents change (with four books, *mrs* was a keyword of one)
- **Inverted index:** `{word: {doc_id: count}}`, so a query only visits the documents that contain its words
- **Ranking:** sum `(1 + math.log(count)) * idf` over the query words; in the test, 10 of 10 right with idf and 4 of 10 without on a query padded with common words
- **Variants:** smoothed idf such as `ln(1 + N / df)` keeps common words from scoring zero (Exercise 1); BM25 is a descendant used in search engines
- **Source conflict:** the Okapi BM25 work is dated 1994 in some sources and 1995 in others

### COMING NEXT

Words by the Numbers 9 is about information and compression: surprise and entropy, Shannon's guessing games, and how much English can be squeezed.

# Words by the Numbers

Chapter 9 · Information and Compression

Words by the Numbers 8 ranked words by how much they tell you about a document. This chapter asks a sharper version of the question: how much does each *letter* tell you, and what is the least space in which English can be written down? The answer is a number measured in bits, it comes from the same counting you have been doing since Words by the Numbers 1, and it explains why a zip file of a novel is so much smaller than the novel. The maths is one logarithm again: the base-2 logarithm, which counts bits.

**WHAT WAS RUN, AND WHAT TO TRUST**

Every Python snippet in this chapter and in the three exercise solutions was executed (Python 3.14.0), and the output in the comments is what Python printed. The four books are plain-text files from Project Gutenberg (eBook numbers 11, 35, 1342 and 1404), which the code downloads once and saves next to the script; you need an internet connection the first time. For the information measures the text is cleaned to 27 symbols, the letters a to z and a single space, so punctuation, capitals and digits are thrown away; numbers for real files with punctuation would differ. The probabilities in the first snippet (such as the 90% chance of rain) are *invented, illustrative* numbers. The compression numbers use the Python standard-library modules zlib, bz2 and lzma at the settings shown, and other compressors would give other figures. The history comes from two web searches that returned summaries of web pages; such fetched pages are often truncated or model-summarised, so treat the wording as a paraphrase. The searches supported the journal, volume, pages and date of Shannon's prediction paper and the date and journal of Huffman's coding paper, and they showed a source conflict about Shannon's final estimate, noted below. Statements marked (check) were not confirmed against a source.

## Surprise, Measured in Bits

Claude Shannon, working at Bell Labs, set out the mathematics of information in his 1948 paper "A Mathematical Theory of Communication" (the year and title are from memory, not from this chapter's searches, so check them). Its starting point is simple. An event that you expected carries little news; an event that you did not expect carries a lot. Shannon measured that news, the **surprise** of an event with probability `p`, as `-log2(p)` bits. A fair coin landing heads has `p = 1/2` and so exactly 1 bit of surprise: the answer to one yes/no question. A specific face of a

fair die has `p = 1/6`, and `-log2(1/6) = 2.585` bits, a little more than the 2 bits that two yes/no questions can settle and a little less than 3.

The **entropy** of a whole distribution is the average surprise, each outcome's surprise weighted by how often it happens: `H = sum of p * -log2(p)`. It is the number of bits you need on average to say which outcome occurred, and it is the quantity the rest of the chapter is about. The first snippet works both ideas by hand on coins, dice and cards.

```
import math

def surprise(p):
    return -math.log2(p)                    # bits of surprise for an event of probabi

def entropy(probs):
    return sum(p * surprise(p) for p in probs if p > 0)

print("surprise of single events (bits):")
for label, p in [("fair coin lands heads", 1/2), ("one specific die face", 1/6),
                 ("one specific card from 52", 1/52), ("rain tomorrow (invented p = 0.9)
    print(f"  {label:34} p {p:.3f}  surprise {surprise(p):.3f}")

print("entropy of whole distributions (bits):")
for label, probs in [("fair coin", [0.5, 0.5]),
                     ("coin that lands heads 90%", [0.9, 0.1]),
                     ("fair die", [1/6] * 6),
                     ("die with a 50% six", [0.1] * 5 + [0.5]),
                     ("26 equally likely letters", [1/26] * 26)]:
    print(f"  {label:28} {entropy(probs):.3f}")
# surprise of single events (bits):
#   fair coin lands heads              p 0.500  surprise 1.000
#   one specific die face              p 0.167  surprise 2.585
#   one specific card from 52          p 0.019  surprise 5.700
#   rain tomorrow (invented p = 0.9)   p 0.900  surprise 0.152
# entropy of whole distributions (bits):
#   fair coin                    1.000
#   coin that lands heads 90%    0.469
#   fair die                     2.585
#   die with a 50% six           2.161
#   26 equally likely letters    4.700
```

| DISTRIBUTION | ENTROPY (BITS) | WHY |
|---|---|---|
| fair coin | 1.000 | two equally likely outcomes, one yes/no question |
| coin that lands heads 90% | 0.469 | mostly predictable, so little news on average |
| fair die | 2.585 | log2 of 6, the most a six-outcome die can have |
| die with a 50% six | 2.161 | lopsided, so less than the fair die |
| 26 equally likely letters | 4.700 | log2 of 26, the ceiling for a 26-letter alphabet |

The pattern to remember: for a fixed number of outcomes, entropy is highest when they are all equally likely (`log2` of the number of outcomes) and falls as the distribution becomes lopsided. English letters are very lopsided. That is the whole reason English can be compressed.

## What Is a Letter Worth?

Words by the Numbers 3 counted how often each letter occurs. Those frequencies are probabilities, so the same formula gives the entropy of English letters. A fixed-length code spends the same effort on every symbol: 27 symbols need 5 bits each (`log2(27) = 4.755`, rounded up because you cannot send part of a bit in a fixed-width code). A **variable-length code** gives short codes to common symbols and long ones to rare ones, as Morse code does with *e* and *q*. David Huffman published a method for building the best such code, in a 1952 paper "A Method for the Construction of Minimum-Redundancy Codes" in *Proceedings of the IRE* (volume 40, issue 9, pages 1098 to 1101, per the search; the search also said he developed it as a student at MIT). The algorithm is short: repeatedly merge the two least frequent symbols into one, and every merge pushes the symbols inside one level deeper in the code. The second snippet builds it from the letter counts of each book and compares it with the entropy.

```
import heapq, math, pathlib, re, urllib.request
from collections import Counter

BOOKS = {"Alice in Wonderland": 11, "The Time Machine": 35,
         "Pride and Prejudice": 1342, "The Federalist Papers": 1404}

def load(number):
    path = pathlib.Path(f"pg{number}.txt")
    if not path.exists():                       # download once, then reuse the saved co
```

```
        url = f"https://www.gutenberg.org/cache/epub/{number}/pg{number}.txt"
        req = urllib.request.Request(url, headers={"User-Agent": "Mozilla/5.0"})
        path.write_bytes(urllib.request.urlopen(req).read())
    raw = path.read_text(encoding="utf-8-sig").replace("\r", "")
    start = raw.index("\n", raw.index("*** START OF")) + 1
    return raw[start:raw.index("*** END OF")]

def clean(text):                                # 27 symbols: a-z and one space
    return re.sub(r"[^a-z]+", " ", text.lower()).strip()

def entropy(counts):
    total = sum(counts.values())
    return -sum(c / total * math.log2(c / total) for c in counts.values())

def huffman_lengths(counts):                    # code length (bits) for each symbol
    heap = [(c, i, [s]) for i, (s, c) in enumerate(counts.items())]
    heapq.heapify(heap)
    depth = {s: 0 for s in counts}
    n = len(heap)
    while len(heap) > 1:
        c1, _, a = heapq.heappop(heap)
        c2, _, b = heapq.heappop(heap)
        for s in a + b:
            depth[s] += 1                       # every merge pushes its symbols one lev
        n += 1
        heapq.heappush(heap, (c1 + c2, n, a + b))
    return depth

print(f"fixed-length code for 27 symbols: 5 bits per character; log2(27) = {math.log2(27
for title, number in BOOKS.items():
    text = clean(load(number))
    letters = Counter(text)
    lengths = huffman_lengths(letters)
    avg = sum(letters[s] * lengths[s] for s in letters) / len(text)
    words = text.split()
    wc = Counter(words)
    print(f"{title}")
    print(f"  characters {len(text):,}   letter entropy {entropy(letters):.3f} bits   Hu
    print(f"  space {lengths[' ']} bits, e {lengths['e']} bits, z {lengths['z']} bits")
    print(f"  word entropy {entropy(wc):.2f} bits per word, {entropy(wc) * len(words) /
# fixed-length code for 27 symbols: 5 bits per character; log2(27) = 4.755
# Alice in Wonderland
#   characters 135,508   letter entropy 4.046 bits   Huffman average 4.083 bits
#   space 2 bits, e 3 bits, z 10 bits
#   word entropy 8.62 bits per word, 1.744 bits per character
```

```
  # The Time Machine
  #   characters 174,220   letter entropy 4.078 bits   Huffman average 4.121 bits
  #   space 2 bits, e 3 bits, z 10 bits
  #   word entropy 9.08 bits per word, 1.714 bits per character
  # Pride and Prejudice
  #   characters 692,488   letter entropy 4.098 bits   Huffman average 4.133 bits
  #   space 3 bits, e 3 bits, z 10 bits
  #   word entropy 9.11 bits per word, 1.691 bits per character
  # The Federalist Papers
  #   characters 1,122,155   letter entropy 4.080 bits   Huffman average 4.116 bits
  #   space 3 bits, e 3 bits, z 10 bits
  #   word entropy 8.90 bits per word, 1.525 bits per character
```

| BOOK | LETTER ENTROPY | HUFFMAN AVERAGE | WORD ENTROPY PER CHARACTER |
|---|---|---|---|
| Alice in Wonderland | 4.046 | 4.083 | 1.744 |
| The Time Machine | 4.078 | 4.121 | 1.714 |
| Pride and Prejudice | 4.098 | 4.133 | 1.691 |
| The Federalist Papers | 4.080 | 4.116 | 1.525 |

Three results. First, the entropy of a single letter in these books is about 4.05 to 4.10 bits, against 4.755 if all 27 symbols were equally likely and 5 for a fixed-length code, so just knowing the letter frequencies saves about 18 to 19% on the fixed code. Second, Huffman's code comes within 0.05 bits of the entropy in every book (it is never more than one bit per symbol above it, a standard result I did not check against a source: check). The space gets 2 or 3 bits, *e* 3 bits and *z* 10, which is the same ordering as the surprise values you will compute in Exercise 1. Third, treating whole words as the symbols gives a much smaller figure, 1.5 to 1.7 bits per character. That number is a trap. It is the entropy of each book's own word counts, and it ignores the cost of describing the vocabulary itself; with thousands of words seen only once, such a plug-in estimate comes out too low, and the word list would have to be sent to the reader as well. It is not a fair estimate of the entropy of English.

## Shannon's Guessing Game

Letter frequencies ignore context: after a *q* the next letter is nearly always *u*, which a single-letter count cannot see. Words by the Numbers 5 used the previous word to predict the next; the same idea works for letters, and Shannon turned it into an experiment. In his 1951 paper "Prediction and Entropy of Printed English" (*Bell System Technical Journal*, volume 30, number 1, pages 50 to 64, per the search), people were shown some text and asked to guess the next letter, then the next, and so on. The number of guesses a person needs says how predictable the text is: if the first guess is usually right, the text carries little information per letter. The sources I could read agree on the broad picture. Statistics over up to eight letters gave about 2.3 bits per letter, and allowing for structure up to about 100 letters pushed the estimate down to around one bit per letter, which means English is about 75% redundant. They do not agree on the exact range: one secondary summary gave a bound of 0.6 to 1.3 bits and another gave 1.0 to 1.5, and a third summary of a replication study was garbled enough that I could not tell the direction of its comparison. I therefore say only "roughly one bit per letter, give or take". Shannon also converted a word-level calculation into about 2.62 bits per letter; that figure comes from the same search and I did not check it in the paper.

The next snippet plays the game with a machine instead of a person. A model trained on text sees the previous `k` characters, ranks all 27 symbols from most to least likely, and we count how far down the list the true next character sits. It also reports the **cross-entropy** in bits per character: the average surprise when the model's probabilities are used on text it was not trained on. For the first table the model is trained on the first half of a book and tested on the second half; for the second, trained on three books and tested on the fourth. The smoothing constant (0.1, added to every count so an unseen next letter does not get probability zero) is my arbitrary choice.

```
import math, pathlib, re, urllib.request
from collections import Counter, defaultdict

BOOKS = {"Alice in Wonderland": 11, "The Time Machine": 35,
         "Pride and Prejudice": 1342, "The Federalist Papers": 1404}
ALPHABET = " abcdefghijklmnopqrstuvwxyz"

def load(number):
    path = pathlib.Path(f"pg{number}.txt")
    if not path.exists():                       # download once, then reuse the saved co
        url = f"https://www.gutenberg.org/cache/epub/{number}/pg{number}.txt"
        req = urllib.request.Request(url, headers={"User-Agent": "Mozilla/5.0"})
        path.write_bytes(urllib.request.urlopen(req).read())
```

```
    raw = path.read_text(encoding="utf-8-sig").replace("\r", "")
    start = raw.index("\n", raw.index("*** START OF")) + 1
    return raw[start:raw.index("*** END OF")]

def clean(text):                                # 27 symbols: a-z and one space
    return re.sub(r"[^a-z]+", " ", text.lower()).strip()

def train(text, k):                             # counts of the next character after eac
    model = defaultdict(Counter)
    for i in range(k, len(text)):
        model[text[i - k:i]][text[i]] += 1
    return model

def cross_entropy(model, text, k, alpha=0.1):   # average bits per character on text the
    bits = 0.0
    for i in range(k, len(text)):
        c = model.get(text[i - k:i], Counter())
        p = (c[text[i]] + alpha) / (sum(c.values()) + alpha * 27)
        bits -= math.log2(p)
    return bits / (len(text) - k)

texts = {title: clean(load(n)) for title, n in BOOKS.items()}
print("bits per character on the second half of each book, model trained on the first ha
for title, text in texts.items():
    half = len(text) // 2
    row = []
    for k in range(4):                          # order k = the previous k characters as
        row.append(f"order {k}: {cross_entropy(train(text[:half], k), text[half:], k):.3
    print(f"  {title:22}", "   ".join(row))

# Shannon's guessing game, played by a machine: guess the most likely character first,
# then the next most likely, and count the guesses needed. Train on three books, test on
print("guesses needed per character (order 2 model, tested on a book it has never seen):
for held_out in BOOKS:
    pool = "".join(t for title, t in texts.items() if title != held_out)
    model = train(pool, 2)
    overall = Counter(pool)
    tally = Counter()
    test = texts[held_out]
    for i in range(2, len(test)):
        c = model.get(test[i - 2:i], Counter())
        order = sorted(ALPHABET, key=lambda ch: (-c[ch], -overall[ch]))
        tally[order.index(test[i]) + 1] += 1
    n = sum(tally.values())
    average = sum(g * v for g, v in tally.items()) / n
```

```
    print(f"  {held_out:22} 1st guess right {tally[1] / n:.1%}, within 3 guesses {sum(ta
# bits per character on the second half of each book, model trained on the first half:
#   Alice in Wonderland    order 0: 4.054   order 1: 3.237   order 2: 2.589   order 3: 2
#   The Time Machine       order 0: 4.077   order 1: 3.281   order 2: 2.683   order 3: 2
#   Pride and Prejudice    order 0: 4.096   order 1: 3.307   order 2: 2.595   order 3: 2
#   The Federalist Papers  order 0: 4.077   order 1: 3.230   order 2: 2.468   order 3: 1
# guesses needed per character (order 2 model, tested on a book it has never seen):
#   Alice in Wonderland    1st guess right 39.3%, within 3 guesses 66.3%, average 3.73 g
#   The Time Machine       1st guess right 38.2%, within 3 guesses 64.5%, average 3.79 g
#   Pride and Prejudice    1st guess right 38.0%, within 3 guesses 64.5%, average 3.86 g
#   The Federalist Papers  1st guess right 39.8%, within 3 guesses 65.8%, average 3.70 g
```

| BOOK (SECOND HALF) | NO CONTEXT | 1 CHARACTER | 2 CHARACTERS | 3 CHARACTERS |
|---|---|---|---|---|
| Alice in Wonderland | 4.054 | 3.237 | 2.589 | 2.262 |
| The Time Machine | 4.077 | 3.281 | 2.683 | 2.325 |
| Pride and Prejudice | 4.096 | 3.307 | 2.595 | 2.041 |
| The Federalist Papers | 4.077 | 3.230 | 2.468 | 1.877 |

Each extra character of context takes a large bite out of the surprise: from about 4.1 bits with no context to between 1.9 and 2.3 with three characters. The guessing results say the same in plainer words. With two characters of context, the machine's first guess is right about 38 to 40% of the time, and it needs 3.7 to 3.9 guesses on average, out of a possible 27. These figures are not comparable with Shannon's one bit, because his readers could use whole words, grammar and meaning over spans of up to about 100 letters, and my model sees at most three characters. Two cautions about my own numbers. The order-3 figures differ a lot between books (1.88 to 2.33); I did not investigate why. And more context is not always better: with a fixed amount of training text, long contexts are mostly unseen and the smoothing does the work, so the curve eventually turns upward (this was not tested here beyond three characters).

## How Much Can English Be Squeezed?

Everything so far is a bound or a model. A compressor is the working version: it turns text into fewer bits and turns them back again. Whatever size it reaches is an upper bound on what the text's entropy rate can be, because it is a scheme that really works. The last snippet runs three

compressors from Python's standard library over each book, and over four controls: the same letters shuffled (destroying everything but letter frequencies), a random string of 27 symbols (nothing to find), and a thousand characters repeated a hundred times (almost nothing but repetition). Sizes are in bits per character of the cleaned text, so 8 would mean no better than storing one byte per character.

```
import bz2, lzma, math, pathlib, random, re, urllib.request, zlib
from collections import Counter

BOOKS = {"Alice in Wonderland": 11, "The Time Machine": 35,
         "Pride and Prejudice": 1342, "The Federalist Papers": 1404}

def load(number):
    path = pathlib.Path(f"pg{number}.txt")
    if not path.exists():                          # download once, then reuse the saved co
        url = f"https://www.gutenberg.org/cache/epub/{number}/pg{number}.txt"
        req = urllib.request.Request(url, headers={"User-Agent": "Mozilla/5.0"})
        path.write_bytes(urllib.request.urlopen(req).read())
    raw = path.read_text(encoding="utf-8-sig").replace("\r", "")
    start = raw.index("\n", raw.index("*** START OF")) + 1
    return raw[start:raw.index("*** END OF")]

def clean(text):                                   # 27 symbols: a-z and one space
    return re.sub(r"[^a-z]+", " ", text.lower()).strip()

def entropy(text):
    counts = Counter(text)
    return -sum(c / len(text) * math.log2(c / len(text)) for c in counts.values())

def bits_per_char(data):
    raw = data.encode("ascii")
    return {"zlib": len(zlib.compress(raw, 9)) * 8 / len(data),
            "bz2": len(bz2.compress(raw, 9)) * 8 / len(data),
            "lzma": len(lzma.compress(raw)) * 8 / len(data)}

print("compressed size in bits per character (a-z and spaces only; 8 bits = no saving ov
for title, number in BOOKS.items():
    text = clean(load(number))
    sizes = bits_per_char(text)
    print(f"  {title:22} letter entropy {entropy(text):.2f}    " + "    ".join(f"{k} {v:.2

# Controls: text with no English structure, and text with a lot of repetition
```

```
random.seed(1)
text = clean(load(11))[:100_000]
controls = {
    "first 100,000 characters of one book": text,
    "the same letters shuffled": "".join(random.sample(text, len(text))),
    "random 27-symbol string": "".join(random.choice(" abcdefghijklmnopqrstuvwxyz") for
    "first 1,000 characters repeated 100 times": text[:1000] * 100,
}
print("controls, bits per character:")
for label, data in controls.items():
    sizes = bits_per_char(data)
    print(f"  {label:42} " + "   ".join(f"{k} {v:.2f}" for k, v in sizes.items()))
print(f"  (log2 of 27 = {math.log2(27):.2f} is the floor for the random string)")
# compressed size in bits per character (a-z and spaces only; 8 bits = no saving over AS
#   Alice in Wonderland    letter entropy 4.05   zlib 2.60   bz2 2.08   lzma 2.31
#   The Time Machine       letter entropy 4.08   zlib 2.85   bz2 2.26   lzma 2.53
#   Pride and Prejudice    letter entropy 4.10   zlib 2.66   bz2 1.87   lzma 2.16
#   The Federalist Papers  letter entropy 4.08   zlib 2.43   bz2 1.69   lzma 1.87
# controls, bits per character:
#   first 100,000 characters of one book       zlib 2.64   bz2 2.15   lzma 2.39
#   the same letters shuffled                  zlib 4.73   bz2 4.46   lzma 4.21
#   random 27-symbol string                    zlib 5.29   bz2 4.84   lzma 4.91
#   first 1,000 characters repeated 100 times  zlib 0.08   bz2 0.10   lzma 0.05
#   (log2 of 27 = 4.75 is the floor for the random string)
```

| BOOK | LETTER ENTROPY | ZLIB | LZMA | BZ2 |
|---|---|---|---|---|
| Alice in Wonderland | 4.05 | 2.60 | 2.31 | 2.08 |
| The Time Machine | 4.08 | 2.85 | 2.53 | 2.26 |
| Pride and Prejudice | 4.10 | 2.66 | 2.16 | 1.87 |
| The Federalist Papers | 4.08 | 2.43 | 1.87 | 1.69 |

| TEXT | WHAT THE COMPRESSOR CAN USE | ZLIB / BZ2 / LZMA (BITS PER CHARACTER) |
|---|---|---|
| first 100,000 characters of one book | letter frequencies, words, repeated phrases | 2.64 / 2.15 / 2.39 |
| the same letters shuffled | letter frequencies only | 4.73 / 4.46 / 4.21 |

| TEXT | WHAT THE COMPRESSOR CAN USE | ZLIB / BZ2 / LZMA (BITS PER CHARACTER) |
|---|---|---|
| random 27-symbol string | nothing | 5.29 / 4.84 / 4.91 |
| 1,000 characters repeated 100 times | exact repetition | 0.08 / 0.10 / 0.05 |

Read the controls first, because they show what the compressors are finding. Shuffling the letters leaves them near the single-letter entropy (about 4.1, and lzma got to 4.21), so the gap between 4.2 and 2.4 on real text is the value of ordering: words, spelling and repeated phrases. The random string cannot be squeezed below its entropy of 4.755 and the compressors do not manage it: zlib needs 5.29 bits per character, more than the plain 5-bit code, because its bookkeeping costs more than it saves when there is nothing to find. At the other extreme, a text that is just repetition shrinks to under a tenth of a bit per character. On real books the three tools land between 1.7 and 2.9 bits per character, that is 64% to 79% smaller than a byte per character. bz2 and lzma beat the order-3 figure for Alice in Wonderland, zlib did not, and all three are a long way from Shannon's one bit, which hints at what is still left in the structure of English, and why modern language models, which predict text from long contexts, are also compressors in disguise (I am stating that last link from general knowledge, not from a source I checked). One more warning on comparing columns: the cross-entropy figures earlier were measured on held-out halves, while the compressors were measured on whole books, so the two groups are close cousins, not a fair race.

| METHOD | CONTEXT IT USES | ALICE IN WONDERLAND (BITS PER CHARACTER) |
|---|---|---|
| fixed-length code | none | 5.00 |
| letter entropy (Huffman, 4.08) | none, just letter counts | 4.05 |
| order-2 model, held-out half | previous 2 characters | 2.59 |
| order-3 model, held-out half | previous 3 characters | 2.26 |
| zlib / lzma / bz2, whole book | repeats over the whole text | 2.60 / 2.31 / 2.08 |
| Shannon's human readers (1951) | meaning and long range | roughly 1 (see the conflict above) |

Common mistakes

Treating a bit as a unit of meaning rather than of surprise (entropy measures unpredictability, not importance: a random string has the highest entropy of all); using `log` instead of `log2` and getting nats or a different scale; including a symbol with probability zero in the sum (`0 * log 0` is undefined, so skip it); testing a model on the text it was trained on, which flatters it badly; quoting one entropy number for "English" without saying the symbols, the cleaning and the context length; trusting a word-level entropy computed from one text's own counts; comparing compressors on different inputs, or on a file too small for the compressor's overhead to be ignored; and forgetting that a smoothing constant or a context length is a choice that moves the answer.

**EXERCISE 1**

For Pride and Prejudice cleaned to a-z and spaces, print the surprise of every symbol, show the three least and three most surprising ones, and confirm that the frequency-weighted average of the surprises is the entropy. Then compare the surprise of *u* anywhere with the surprise of *u* straight after *q*. What does the difference tell you about context?

View solution

**EXERCISE 2**

Build a real Huffman code for the letters of Alice in Wonderland, encode the first 20,000 characters into a string of 0s and 1s, decode it back, and check you recover the original. Report the bits per character and compare it with the entropy and with a fixed 5-bit code. Why can the code be decoded without separators between symbols?

View solution

**EXERCISE 3**

Play Shannon's guessing game with a machine that has read only the other three books. For context lengths 0 to 3 print the average number of guesses needed on Alice in Wonderland, then for context length 2 print the number of guesses for each character of a 60-character passage. Where do the long runs of 1s occur, and which letters cost the most guesses?

View solution

### QUICK REFERENCE

- **Surprise:** `-math.log2(p)` bits for an event of probability p; a fair coin is 1 bit, a fair die face 2.585
- **Entropy:** the average surprise, `sum(p * -math.log2(p))`; highest when outcomes are equally likely (`log2(n)`), lower when lopsided
- **Letter entropy:** about 4.05 to 4.10 bits per symbol in the four books (27 symbols, uniform would be 4.755, a fixed code 5)
- **Huffman code:** short codes for common symbols, built by repeatedly merging the two rarest; within 0.05 bits of the entropy here
- **Context helps:** held-out cross-entropy fell from about 4.1 (no context) to 1.9 to 2.3 bits (three characters); an order-2 model needed 3.7 to 3.9 guesses per character
- **Compressors:** zlib, lzma and bz2 reached 1.7 to 2.9 bits per character on cleaned book text; shuffled letters stayed near 4.2 to 4.7, random symbols could not be squeezed, and zlib made them bigger than 5 bits
- **Shannon 1951:** human guessing gave an estimate of roughly one bit per letter (sources conflict on the range, 0.6 to 1.3 in one summary and 1.0 to 1.5 in another)
- **Pitfalls:** test on held-out text, state the symbol set and context length, and do not use a word entropy computed from a text's own counts as the entropy of English

### COMING NEXT

Words by the Numbers 10 is the capstone, a text detective: combining the tools to identify the language, author and topic of mystery passages.

# Words by the Numbers

Chapter 10 · Capstone: A Text Detective

A stranger hands you a page of text with no title and no name. Can you say what language it is in, who probably wrote it, and what it is about? This last chapter puts the course to work on exactly that. Each question gets one tool you already own: letter counts from Words by the Numbers 3, the author comparison from Words by the Numbers 6, and the keyword ranking from Words by the Numbers 8. The new skill is not another formula. It is knowing how far to trust each verdict, which is why the chapter spends as much time on the failures as on the successes.

**WHAT WAS RUN, AND WHAT TO TRUST**

Every Python snippet in this chapter and in the three exercise solutions was executed (Python 3.14.0), and the output in the comments is what Python printed. The texts are plain-text files from Project Gutenberg, which the code downloads once and saves next to the script, so you need an internet connection the first time. The books are identified by their eBook numbers: for language, English 1342 and 11 and 12 and 36, French 17489 and 2650, German 2229 and 22367 and 2407, Spanish 2000 and 49836, Italian 1012 and 45334; for authors, 1342 and 158 (Austen), 11 and 12 (Carroll), 35 and 36 (Wells), 1661 and 2852 (Doyle), 98 and 730 (Dickens), with 1260 as a stranger to the system. Every test passage comes from a different book than the one the profile was trained on, so the accuracy figures are honest, but they use only these few books and one random seed, and other books or seeds would give other numbers. The methods are simplified: my language test uses letter-pair profiles with cosine similarity, and my author test is a simple version of the Delta measure with my own choices (2,000-word reference chunks, the commonest words of the five training books). The history comes from two web searches that returned summaries of web pages, which are often truncated or model-summarised, so treat the wording as a paraphrase. The searches supported the citations given below and the general descriptions of the two methods; they did not let me check the original papers. Statements marked (check) were not confirmed against a source.

## Case 1: What Language Is This?

Language identification is the easiest case, and it is old. A 1994 paper by William Cavnar and John Trenkle, "N-Gram-Based Text Categorization" (presented at the SDAIR-94 symposium in Las Vegas, pages 161 to 175, per the search), built a profile of the most frequent short letter sequences for each language and gave a new text to the language whose profile it matched

best. Secondary summaries report accuracy of about 99.8% on postings of at least 300 bytes, using the 400 most frequent sequences, and 99.3% on shorter ones; I did not check those figures in the paper, and the search did not say which sequence lengths they used (check). The idea is the one from Words by the Numbers 3, that every language has a letter profile, plus the idea from Words by the Numbers 4, that pairs of neighbours say more than single items.

The first snippet trains one profile per language on one book, then tests passages of 20 to 300 characters taken at random from different books. It does this twice: once with single letters as the features and once with letter pairs. The score is cosine similarity, which measures how closely two lists of frequencies point the same way (1 means identical shape, 0 means nothing in common). Each result below is out of 500 passages, 100 per language.

```
import math, pathlib, random, re, urllib.request
from collections import Counter

TRAIN = {"English": 1342, "French": 17489, "German": 2229, "Spanish": 2000, "Italian": 1(
TEST = {"English": [12, 36], "French": [2650], "German": [22367, 2407],
        "Spanish": [49836], "Italian": [45334]}      # different books from the training

def load(number):
    path = pathlib.Path(f"pg{number}.txt")
    if not path.exists():                            # download once, then reuse the saved co
        url = f"https://www.gutenberg.org/cache/epub/{number}/pg{number}.txt"
        req = urllib.request.Request(url, headers={"User-Agent": "Mozilla/5.0"})
        path.write_bytes(urllib.request.urlopen(req).read())
    raw = path.read_text(encoding="utf-8-sig").replace("\r", "")
    start = raw.index("\n", raw.index("*** START OF")) + 1
    return raw[start:raw.index("*** END OF")]

def clean(text):                                     # lower-case letters (accents kept) and
    return re.sub(r"[\W\d_]+", " ", text.lower()).strip()

def profile(text, n):                                # relative frequency of each n-letter se
    grams = Counter(text[i:i + n] for i in range(len(text) - n + 1) if " " not in text[i
    total = sum(grams.values())
    return {g: c / total for g, c in grams.items()}

def cosine(a, b):
    dot = sum(v * b.get(g, 0) for g, v in a.items())
    return dot / (math.sqrt(sum(v * v for v in a.values())) * math.sqrt(sum(v * v for v

def guess(passage, profiles, n):
```

```
    p = profile(passage, n)
    return max(profiles, key=lambda lang: cosine(p, profiles[lang]))

random.seed(1)
trained = {lang: clean(load(n)) for lang, n in TRAIN.items()}
pools = {lang: [clean(load(n)) for n in nums] for lang, nums in TEST.items()}
print("training text (characters):", {lang: f"{len(t):,}" for lang, t in trained.items()}
for n in (1, 2):
    profiles = {lang: profile(t, n) for lang, t in trained.items()}
    print(f"profile of {n}-letter sequences; accuracy on 100 random passages per languag
    for length in (20, 50, 100, 300):
        right = total = 0
        for lang, books in pools.items():
            for _ in range(100):
                book = random.choice(books)
                start = random.randrange(len(book) - length)
                right += guess(book[start:start + length], profiles, n) == lang
                total += 1
        print(f"  {length:4} characters: {right / total:.1%}")
profiles = {lang: profile(t, 1) for lang, t in trained.items()}
for letter in "eaznwkyçñ":
    print(f"  {letter}: " + "  ".join(f"{lang[:2]} {profiles[lang].get(letter, 0):.2%}" 
# training text (characters): {'English': '692,498', 'French': '632,388', 'German': '178
# profile of 1-letter sequences; accuracy on 100 random passages per language:
#      20 characters: 57.0%
#      50 characters: 75.6%
#     100 characters: 86.4%
#     300 characters: 96.0%
# profile of 2-letter sequences; accuracy on 100 random passages per language:
#      20 characters: 75.4%
#      50 characters: 91.0%
#     100 characters: 97.4%
#     300 characters: 99.0%
#   e: En 12.87%  Fr 14.76%  Ge 15.48%  Sp 13.53%  It 11.55%
#   a: En 7.77%  Fr 8.17%  Ge 4.90%  Sp 11.79%  It 10.54%
#   z: En 0.17%  Fr 0.19%  Ge 1.03%  Sp 0.40%  It 0.46%
#   n: En 7.03%  Fr 6.76%  Ge 9.53%  Sp 6.61%  It 6.59%
#   w: En 2.26%  Fr 0.00%  Ge 1.83%  Sp 0.00%  It 0.00%
#   k: En 0.60%  Fr 0.00%  Ge 1.10%  Sp 0.00%  It 0.00%
#   y: En 2.35%  Fr 0.34%  Ge 0.03%  Sp 1.53%  It 0.00%
#   ç: En 0.00%  Fr 0.07%  Ge 0.00%  Sp 0.00%  It 0.00%
#   ñ: En 0.00%  Fr 0.00%  Ge 0.00%  Sp 0.26%  It 0.00%
```

| CHARACTERS IN THE PASSAGE | SINGLE LETTERS | LETTER PAIRS |
|---|---|---|
| 20 | 57.0% | 75.4% |
| 50 | 75.6% | 91.0% |
| 100 | 86.4% | 97.4% |
| 300 | 96.0% | 99.0% |

Three things stand out. Pairs beat single letters at every length, because a pair like `th` or `qu` carries more of a language's habits than a letter count does. Both improve with more text, as Words by the Numbers 1 would lead you to expect: more data, less luck. And the letter table at the end of the output shows why a single-letter profile works at all, and where it is weak. The letters `w` and `k` are absent from the French, Spanish and Italian profiles, `z` is six times commoner in German than in English (1.03% against 0.17%), and `ñ` and `ç` identify Spanish and French almost alone. But `e` and `n` are everywhere, and the pairs add the rest. The training books are modest (the German one is only 179 thousand characters), and the test is five languages that all use the Latin alphabet; languages written in other scripts would be easier to tell apart, and closely related languages harder.

| LANGUAGE | WHAT A LETTER PROFILE PICKS UP | CAUTION |
|---|---|---|
| English | w and y, and pairs such as th | accuracy may depend on how old the test text is (check) |
| French | c with a cedilla, accented e, no w or k | an English quotation inside a French novel confuses it |
| German | z, k, w, and clusters such as sch | small training book here, so the profile is noisier |
| Spanish | n with a tilde, a and o endings | most often confused with Italian in Exercise 1 |
| Italian | a, i and o endings, few k, w or y | most often confused with Spanish in Exercise 1 |

## Case 2: Who Wrote This?

Authorship is harder, because every author writes the same language. Words by the Numbers 6 used the habit that is hardest to fake: how often a writer uses small function words such as *the*, *of* and *but*. Here is the measure in its general form. Standardise each word's rate against a reference collection (subtract the mean, divide by the standard deviation) so that all words count equally, then take the average absolute difference between the passage's standardised rates and

each author's. This is Burrows' Delta, from John Burrows, "'Delta': a Measure of Stylistic Difference and a Guide to Likely Authorship", *Literary and Linguistic Computing*, volume 17, number 3, 2002, pages 267 to 287, per the search. The search also reported a reviewer's note that it works least well on texts of a genre unlike the rest of an author's work, and a 2004 test of the measure by David Hoover in the same journal (volume 19, pages 453 to 475). My version standardises against 2,000-word chunks of the five training books, which is a simplification of Burrows' reference set.

The second snippet trains on one book by each of five authors and tests on a different book by each. It varies two knobs: how many of the commonest words are used, and how long the passage is. It ends with a deliberate trap, a sixth author the system has never heard of, to see whether Delta can recognise a stranger.

```
import pathlib, random, re, statistics, urllib.request
from collections import Counter

TRAIN = {"Austen": 1342, "Carroll": 11, "Wells": 35, "Doyle": 1661, "Dickens": 98}
TEST = {"Austen": 158, "Carroll": 12, "Wells": 36, "Doyle": 2852, "Dickens": 730}
OUTSIDER = 1260                                  # a different author, not among the can

def load(number):
    path = pathlib.Path(f"pg{number}.txt")
    if not path.exists():                        # download once, then reuse the saved co
        url = f"https://www.gutenberg.org/cache/epub/{number}/pg{number}.txt"
        req = urllib.request.Request(url, headers={"User-Agent": "Mozilla/5.0"})
        path.write_bytes(urllib.request.urlopen(req).read())
    raw = path.read_text(encoding="utf-8-sig").replace("\r", "")
    start = raw.index("\n", raw.index("*** START OF")) + 1
    return raw[start:raw.index("*** END OF")]

def words(text):
    return re.findall(r"[a-z']+", text.lower())

def rates(tokens, vocab):                        # occurrences per 1,000 words for each w
    c = Counter(tokens)
    return [1000 * c[w] / len(tokens) for w in vocab]

random.seed(7)
train = {a: words(load(n)) for a, n in TRAIN.items()}
test = {a: words(load(n)) for a, n in TEST.items()}
outsider = words(load(OUTSIDER))
```

```
def build(top_n):
    vocab = [w for w, _ in Counter(sum(train.values(), [])).most_common(top_n)]
    chunks = [rates(t[i:i + 2000], vocab) for t in train.values() for i in range(0, len(t
    mean = [statistics.fmean(col) for col in zip(*chunks)]
    sd = [statistics.pstdev(col) or 1 for col in zip(*chunks)]
    z = lambda r: [(x - m) / s for x, m, s in zip(r, mean, sd)]
    return vocab, {a: z(rates(t, vocab)) for a, t in train.items()}, z

def delta(za, zb):
    return sum(abs(x - y) for x, y in zip(za, zb)) / len(za)

def attribute(passage, vocab, authors, z):
    zp = z(rates(passage, vocab))
    scores = {a: delta(zp, za) for a, za in authors.items()}
    best = min(scores, key=scores.get)
    return best, scores[best]

print("accuracy, 50 random passages from a different book by each of 5 authors (chance 2
for top_n in (10, 30, 100, 300):
    vocab, authors, z = build(top_n)
    row = []
    for length in (200, 500, 1000, 2000):
        right = total = 0
        for a, t in test.items():
            for _ in range(50):
                s = random.randrange(len(t) - length)
                right += attribute(t[s:s + length], vocab, authors, z)[0] == a
                total += 1
        row.append(f"{length} words {right / total:.0%}")
    print(f"  {top_n:3} commonest words:  " + "   ".join(row))

vocab, authors, z = build(100)
known, strangers = [], []
for a, t in test.items():
    for _ in range(50):
        s = random.randrange(len(t) - 1000)
        known.append(attribute(t[s:s + 1000], vocab, authors, z)[1])
for _ in range(50):
    s = random.randrange(len(outsider) - 1000)
    strangers.append(attribute(outsider[s:s + 1000], vocab, authors, z))
cut = sorted(known)[int(0.95 * len(known))]
print(f"best Delta for known authors: median {statistics.median(known):.2f}, 95th percen
print(f"best Delta for the outsider:  median {statistics.median(d for _, d in strangers)
      f"{sum(d > cut for _, d in strangers)} of 50 passages beyond the cut-off;",
```

```
      "blamed on:", dict(Counter(a for a, _ in strangers)))
# accuracy, 50 random passages from a different book by each of 5 authors (chance 20%):
#    10 commonest words:  200 words 46%   500 words 48%   1000 words 64%   2000 words 68%
#    30 commonest words:  200 words 66%   500 words 71%   1000 words 84%   2000 words 84%
#   100 commonest words:  200 words 66%   500 words 81%   1000 words 88%   2000 words 94%
#   300 commonest words:  200 words 68%   500 words 81%   1000 words 94%   2000 words 99%
# best Delta for known authors: median 0.93, 95th percentile 1.21
# best Delta for the outsider:  median 0.95; 2 of 50 passages beyond the cut-off; blamed
```

| COMMONEST WORDS USED | 200 WORDS | 500 WORDS | 1000 WORDS | 2000 WORDS |
|---|---|---|---|---|
| 10 | 46% | 48% | 64% | 68% |
| 30 | 66% | 71% | 84% | 84% |
| 100 | 66% | 81% | 88% | 94% |
| 300 | 68% | 81% | 94% | 99% |

Chance is 20%, so even ten words and 200 words of text do far better than guessing, and both more words and more text help. With 300 words and a 2,000-word passage the score is 99%. Each cell is only 250 trials with one seed, so differences of a few points between cells are not reliable. The last two lines are the sobering part. For passages by the five known authors the best Delta has a median of 0.93 and a 95th percentile of 1.21. For passages by the stranger the median is 0.95, practically the same, and only 2 of 50 pass the 1.21 cut-off, about the 5% you would expect from known authors by construction. The stranger was blamed on Dickens 20 times, Doyle 16, Wells 12 and Austen 2. Delta always names someone. It measures which candidate is nearest, and nearest is not the same as right.

| QUESTION YOU CAN ANSWER | QUESTION YOU CANNOT ANSWER FROM THE NUMBERS |
|---|---|
| Which of these five authors is this passage closest to? | Is the real author one of the five? |
| Does more text or more words make it more accurate? | How sure is the system about this one passage (the best distance alone does not say)? |
| Which pair of authors does the system mix up most (Exercise 2)? | Why: topic, period and genre all leave traces in function-word rates |

## Case 3: What Is It About?

For topic, Words by the Numbers 8 gave the tool: TF-IDF, which scores a word highly when it is frequent in the passage but rare across the collection. The third snippet is the full detective. It takes six mystery passages of 1,000 words, from books the profiles were not trained on (one is by an author outside the candidate list). For each it works out the language from the first 300 characters, names the nearest author if the language is English (flagging a Delta above the 1.21 cut-off from the previous snippet), and lists the five top TF-IDF keywords, measured against 500-word pieces of the books on the shelf for that language. The true answers are printed last.

```
import math, pathlib, random, re, statistics, urllib.request
from collections import Counter

LANG_TRAIN = {"English": 1342, "French": 17489, "German": 2229, "Spanish": 2000, "Italia
AUTHORS = {"Austen": 1342, "Carroll": 11, "Wells": 35, "Doyle": 1661, "Dickens": 98}
MYSTERIES = {"A": (158, "Emma, Austen"), "B": (2852, "The Hound of the Baskervilles, Doy
             "C": (2650, "Du cote de chez Swann, Proust"), "D": (22367, "Die Verwandlung
             "E": (36, "The War of the Worlds, Wells"), "F": (1260, "Jane Eyre, Bronte (
SHELF = {"English": [1342, 11, 35, 1661, 98, 158, 12, 36, 2852, 730, 1260],
         "French": [17489, 2650], "German": [2229, 22367, 2407]}   # books used to measu

def load(number):
    path = pathlib.Path(f"pg{number}.txt")
    if not path.exists():                          # download once, then reuse the saved co
        url = f"https://www.gutenberg.org/cache/epub/{number}/pg{number}.txt"
        req = urllib.request.Request(url, headers={"User-Agent": "Mozilla/5.0"})
        path.write_bytes(urllib.request.urlopen(req).read())
    raw = path.read_text(encoding="utf-8-sig").replace("\r", "")
    start = raw.index("\n", raw.index("*** START OF")) + 1
    return raw[start:raw.index("*** END OF")]

words = lambda text: re.findall(r"[^\W\d_]+(?:'[^\W\d_]+)?", text.lower())

# 1. language: cosine similarity of letter-pair profiles (as in the language snippet)
def pairs(text):
    t = " ".join(words(text))
    c = Counter(t[i:i + 2] for i in range(len(t) - 1) if " " not in t[i:i + 2])
    return {g: v / sum(c.values()) for g, v in c.items()}
def cosine(a, b):
    return sum(v * b.get(g, 0) for g, v in a.items()) / (math.sqrt(sum(v * v for v in a.
lang_profiles = {lang: pairs(load(n)) for lang, n in LANG_TRAIN.items()}
```

```
def language(passage):
    p = pairs(passage)
    return max(lang_profiles, key=lambda lang: cosine(p, lang_profiles[lang]))

# 2. author: Delta over the 100 commonest words (as in the author snippet)
rates = lambda tokens, vocab: [1000 * Counter(tokens)[w] / len(tokens) for w in vocab]
train = {a: words(load(n)) for a, n in AUTHORS.items()}
vocab = [w for w, _ in Counter(sum(train.values(), [])).most_common(100)]
chunks = [rates(t[i:i + 2000], vocab) for t in train.values() for i in range(0, len(t) -
mean = [statistics.fmean(col) for col in zip(*chunks)]
sd = [statistics.pstdev(col) or 1 for col in zip(*chunks)]
z = lambda r: [(x - m) / s for x, m, s in zip(r, mean, sd)]
author_z = {a: z(rates(t, vocab)) for a, t in train.items()}
def author(tokens):
    zp = z(rates(tokens, vocab))
    scores = {a: sum(abs(x - y) for x, y in zip(zp, za)) / len(zp) for a, za in author_z
    best = min(scores, key=scores.get)
    return best, scores[best]

# 3. topic: TF-IDF keywords, with rarity measured over 500-word pieces of the shelf for
pieces = {lang: [set(t[i:i + 500]) for t in map(lambda n: words(load(n)), books) for i i
          for lang, books in SHELF.items()}
def keywords(tokens, lang, k=5):
    tf, docs = Counter(tokens), pieces[lang]
    score = {w: tf[w] * math.log(len(docs) / (1 + sum(w in d for d in docs))) for w in t
    return sorted(score, key=score.get, reverse=True)[:k]

random.seed(3)
cut = 1.21                                      # 95th percentile of best Delta for kno
for label, (number, truth) in MYSTERIES.items():
    tokens = words(load(number))
    s = random.randrange(len(tokens) - 1000)
    passage = tokens[s:s + 1000]
    lang = language(" ".join(passage)[:300])
    line = f"{label}: language {lang}"
    if lang == "English":
        who, d = author(passage)
        line += f"; author {who} (Delta {d:.2f}{', above the cut-off' if d > cut else ''
    print(line + f"; keywords {', '.join(keywords(passage, lang))}")
print()
for label, (number, truth) in MYSTERIES.items():
    print(f"  {label} was really: {truth}")
# A: language English; author Austen (Delta 0.91); keywords weston, praise, taylor, ball
# B: language English; author Doyle (Delta 0.86); keywords charles, sir, tracey, coombe,
# C: language French; keywords princesse, basin, cambremer, général, euphonique
```

```
# D: language German; keywords prokurist, gregor, vorzimmer, geschäft, prokuristen
# E: language English; author Wells (Delta 1.05); keywords strand, brother, wellington,
# F: language English; author Doyle (Delta 0.83); keywords briggs, rochester, governess,
#
#   A was really: Emma, Austen
#   B was really: The Hound of the Baskervilles, Doyle
#   C was really: Du cote de chez Swann, Proust
#   D was really: Die Verwandlung, Kafka
#   E was really: The War of the Worlds, Wells
#   F was really: Jane Eyre, Bronte (not a candidate)
```

| PASSAGE | LANGUAGE | AUTHOR | KEYWORDS (TOP THREE) | VERDICT |
|---|---|---|---|---|
| A | English | Austen (Delta 0.91) | weston, praise, taylor | correct |
| B | English | Doyle (0.86) | charles, sir, tracey | correct |
| C | French | not attempted | princesse, basin, cambremer | language correct |
| D | German | not attempted | prokurist, gregor, vorzimmer | language correct |
| E | English | Wells (1.05) | strand, brother, wellington | correct |
| F | English | Doyle (0.83) | briggs, rochester, governess | wrong: a candidate was blamed for a stranger |

Every language verdict is right, and so are the three author verdicts on passages by candidate authors; the stranger is the lesson. Passage F is by an author who is not on the list, and the detective named Doyle with a Delta of 0.83, which is below the median of 0.93 for passages by the real candidates, so no cut-off could have caught it. Yet the keywords for F (*rochester*, *governess*, *eyre*) are the strongest clue in the whole report, since a character name is a fingerprint of one book. A good detective uses the tools against each other. A keyword list full of proper names says the topic evidence is really about the story, not the style; a Delta that is merely adequate says to hold the verdict loosely; and language, the most reliable of the three, comes first because the other two depend on it. Notice also why the keywords are mostly

names: with a shelf this small, a character who appears in one book is rare everywhere else, so TF-IDF loves characters. A bigger shelf with many books on the same subject would favour subject words instead (my expectation, not tested here).

The last table lines the three tools up so you can choose between them. The reliability column summarises what this chapter measured, not a general rule.

| TOOL | NEEDS | RELIABLE WHEN | FAILS WHEN |
|---|---|---|---|
| letter-pair profile (language) | one book per language, 50 to 300 characters | the language is among those trained | related languages, very short text, mixed-language text |
| Delta on common words (author) | a book or two per candidate, 500 or more words | the true author is among the candidates | the author is a stranger, the genre differs, the passage is short |
| TF-IDF keywords (topic) | a shelf of other texts in the same language | the text has distinctive vocabulary | the shelf is small (names dominate) or the text is generic |

Common mistakes

Testing on the same book you trained on (the numbers look wonderful and mean nothing); reading a nearest-author answer as proof when the real author might not be on the list; quoting one accuracy figure without the passage length and the number of words used; trusting a single random seed for small differences between cells; leaving accented letters in some profiles and stripping them in others; training on a text with a lot of foreign quotation or editorial notes in it; feeding the detective a passage too short for the method (Exercise 3 shows what 30 words does); treating TF-IDF keywords that are character names as the topic of a whole genre; and forgetting that a cut-off taken from known data will always flag about the percentage you chose.

**EXERCISE 1**

How short can a language test be? Using letter-pair profiles, test 100 random passages per language at lengths of 10, 20, 50, 100 and 200 characters, print a table of accuracy by language and length, and report the most common mistake at 20 characters. Which two languages are confused most, and why might that be?

View solution

**EXERCISE 2**

Who gets confused with whom? Build the Delta profiles from the 100 commonest words, test 100 random 1,000-word passages from a different book by each of the five authors, and print a confusion table with the true author in the rows and the named author in the columns. Name the most frequent mistake and suggest, with a reason, why that pair is close.

View solution

**EXERCISE 3**

Make the detective safe to hand to someone else: it must not crash when the language has no shelf for keywords. Run it on three new passages: 1,000 words of a poem, 1,000 words of a Spanish novel, and just 30 words of a book the author profiles do know. Which parts of each verdict would you trust, and which would you mark as uncertain?

View solution

## QUICK REFERENCE

- **Language:** a letter-pair profile per language plus cosine similarity reached 75.4% at 20 characters, 91.0% at 50 and 99.0% at 300 here; single letters reached 57.0%, 75.6% and 96.0%
- **Author:** a simple Delta (mean absolute difference of standardised word rates) reached 88% with the 100 commonest words and 1,000-word passages, and 99% with 300 words and 2,000-word passages, on five authors (chance 20%)
- **Topic:** TF-IDF keywords against a shelf of other texts; on a small shelf they are mostly character names
- **The stranger problem:** Delta always names a candidate; the outsider's best Delta (median 0.95) looked like the known authors' (median 0.93)
- **Order of work:** language first (the others depend on it), then author, then topic, then check them against each other
- **Honest testing:** train and test on different books, report the passage length, the number of features and the seed, and keep small samples in mind
- **History:** Cavnar and Trenkle, SDAIR-94 (1994), for n-gram language profiles; Burrows, *Literary and Linguistic Computing* 17(3), 2002, for Delta (both as summarised by the search; check)

## WHERE TO GO FROM HERE

You have now built, in plain Python, most of the classic toolkit for treating text as numbers: counting (Words by the Numbers 1), the rank-frequency curve (Words by the Numbers 2), letter profiles (Words by the Numbers 3), word pairs (Words by the Numbers 4), generators (Words by the Numbers 5), style (Words by the Numbers 6), readability (Words by the Numbers 7), keywords (Words by the Numbers 8) and information (Words by the Numbers 9). Three suggestions for what to try next, all my own suggestions rather than anything checked against a source. First, scale up: repeat the detective with ten books per author and ten languages, and see which weak spots shrink and which do not. Second, borrow a library: packages for natural-language processing and machine learning offer ready-made versions of these ideas (their names and features change often, so read their current documentation), and having built the simple versions yourself you can now judge what they do. Third, pick a mystery of your own: a diary, a set of old letters, your own emails over ten years, a set of speeches. The tools do not change, but every new collection teaches you something about their limits that no book can, and the habit this course has tried to build, testing on held-out text and being honest about the failures, is the part that carries over to everything else.