Fraunhofer

Preparing Data for Large Language Models

Lessons from training foundation models

Dr.-Ing. Nicolo' Brandizzi
Fraunhofer IAIS, NetMedia

Table of contents

Introduction 2
- Learning objectives 2
- Why this matters 2
- How this is organized 3

Part 0 Basic concepts 3
- Where data processing sits 3
- Kinds of data 3
- Data quality 4
- Filters and signals 4
- Compute and time 4

Part 1 The pipeline 4
- Not all text is equal 5
- The fork: curated vs web 5
- Curated pipeline 6
- Web pipeline 9
- Both paths converge 13
- Recap 14

Part 2 Working at scale 14
- Scale changes the problem 14
- Every decision is a compute decision 15
- Deduplication is the expensive step 16
- Compute time versus wall-clock time 17
- Recap 18

Part 3 The quality question 18
- Surface heuristics: cheap, transparent, blind 18
- Asking a model to judge 20
- The cost in euros 20
- Recap 22

Part 4 Cheap proxies for good text 22
- Bad is easy to spot, good is not 23
- Every quality signal is a cheap proxy 23
- A wider family of proxies 26
- What "good" still leaves open 26
- Recap 26

Closing 27
Bibliography 27

Introduction

Generative AI runs on data. A large language model learns everything it knows from a vast body of text, so before any training can start that text has to be gathered and prepared, and preparing it is one of the hardest and least visible parts of building the model. This material is a practical, self-paced introduction to how that preparation is done, written for practitioners who can already code but are not data specialists; plan on about an hour to read it, longer if you stop to work the exercises. Specifically we treat **data processing**: the work of turning raw text into something a model can learn from. It is one neighbor in a larger family that includes data management and data governance. Most of that data is *training* data, but the same care applies to the smaller *evaluation* and *fine-tuning* sets a model also needs.

What follows comes from our experience training large language models at Fraunhofer IAIS, across several projects: OpenGPT-X, OpenEuroLLM, and SOOFI. OpenGPT-X produced Teuken, a family of open models trained on all 24 official languages of the European Union. The lessons are drawn from building real training data and generalized so they transfer to any large-scale text pipeline.

> **i** Prerequisites
>
> You should be comfortable **reading code**: the examples are short Python snippets, there to make an idea concrete, never for you to write or run. A **general sense of what a language model is** (text in, next-token prediction out) is enough on the modeling side. No prior experience with data pipelines, web crawling, or natural-language processing is assumed; building that understanding is the point of this material.

Learning objectives

By the end of this material you will be able to:

- Explain why **data processing** matters for what a model can do.
- Define the **basic concepts** of data processing, starting with *data quality* and *filtering* (Part 0).
- Trace how raw text, whether **curated** or scraped from the open **web**, becomes clean, uniform training data (Part 1).
- Reason about why **scale** makes compute the binding constraint rather than logic alone (Part 2).
- Distinguish the two things a pipeline produces at once, the **data** and the **traceable record** of how it was made, and say why the record matters as much as the data (Parts 1–2).
- Explain why **quality signals** are proxies for judgments too expensive to run directly, and why good text is approximated rather than measured (Parts 3–4).

Each part opens with its own, more specific objectives.

> **i** How this material works
>
> In this material you will meet short "predict, then reveal" checks and small reasoning tasks that mark where it is your turn to think.

Why this matters

A language model is a product of its training data. It has no knowledge of the world beyond the text it was shown, so the size, the languages, and the quality of that text set the ceiling on what the model can do. Get the data wrong and no amount of clever training recovers it. This is also where transparency is thinnest: how a model's data was processed is rarely reported in any detail, which is part of why the work is worth teaching.

None of that text arrives ready to use. It has to be found at the scale of hundreds of terabytes, cleaned, labeled, scored for quality, and deduplicated. You may never run a pipeline this size yourself, but the lesson still applies: the datasets you download and train on were shaped by exactly these steps, and knowing how they were made tells you what you can trust them for.

Two questions run underneath all of it and shape the rest of the material: where does the enormous amount of suitable text a model needs come from, and what does *suitable* even mean?

How this is organized

Part 0 fixes the basic vocabulary. From there the material follows the data through four parts. **Part 1** builds the pipeline: how raw text, whether curated or web, becomes a single clean training format. **Part 2** turns to scale, where compute cost becomes the binding constraint. **Part 3** confronts the question the pipeline leaves open: how to judge whether text is any good. **Part 4** answers it as far as it can be answered: every quality signal is a cheap proxy for a judgment too costly to run directly, and what makes text *good* is approximated, never defined outright. A short close returns to the question we started from, and what the four parts have answered.

PART 0

Basic concepts

Before following data through the pipeline, it helps to fix a little vocabulary. If you are already comfortable with these terms, skip ahead to Part 1.

By the end of Part 0, you will be able to:

- Place **data processing** among its neighbors, data management and data governance.
- Name the **kinds of data** a model uses: training, evaluation, and fine-tuning.
- Define **data quality**, and explain why it has no single fixed definition for training text.
- Describe what a **filter** does, and how it differs from the **signal** it acts on.
- Distinguish **wall-clock time** from **compute time**, the two budgets every large run lives within.

Where data processing sits

Three terms sit close together and are easy to confuse. They name different jobs over the same data. Definitions vary between frameworks, but a useful working split is:

- **Data governance** sets the rules: who may use which data, for what, under which policies, and who is accountable. It is about decisions and oversight[1].
- **Data management** carries those rules out across the data's life: storing, cataloging, securing, and keeping it available and correct.
- **Data processing** is one step inside management: transforming raw data into a usable form. For us, that means turning messy raw text into clean training data.

Put simply, governance decides *what is allowed and required*, management *keeps the data*, and processing *transforms it*. This material lives in the last of the three. It touches the other two only where they constrain the processing, for example when a record must stay traceable so a fault can be assigned later.

Kinds of data

Not all data plays the same role. The bulk is **training data**, the text a model learns from and the subject of this material. Smaller sets serve other purposes: **evaluation data** measures how well a

[1]Governance is also where shared standards live. In the EU, it is widely organized around the **FAIR principles**: the requirement that data be Findable, Accessible, Interoperable, and Reusable.

trained model performs, and **fine-tuning data** adapts it to a particular task or style. Evaluation and fine-tuning sets are far smaller in volume than the training corpus, though the same quality concerns apply to all three.

Data quality

There is no single number that captures whether text is "good." The closest standard definition comes from ISO/IEC 25012, which describes data quality as the degree to which the characteristics of data satisfy stated and implied needs under specified conditions [1]. An older and more intuitive phrasing calls quality *fitness for use*[2]: data is high quality when it serves the purpose you have for it.

That phrasing carries a warning for everything that follows. Quality is defined against a use, and the use here is training a capable language model, so what counts as "good text" depends on a downstream outcome no filter can observe directly. Parts 3 and 4 return to this.

Filters and signals

Two words appear throughout the pipeline and are worth separating now.

A **signal** measures something about a piece of text: the share of characters that are symbols, the average number of words per line, a language-detection confidence, a model's quality score. A signal is a measurement.

A **filter** is the decision that acts on a signal. It sets a threshold and keeps or drops each document accordingly. A signal tells you a document is 60% symbols; a filter is the rule "drop anything above 30%."

Keeping the two ideas distinct matters later, when we ask what a filter can and cannot know. A filter is only ever as good as the signal beneath it, and a signal measures the surface of the text; what the words actually mean stays out of its reach.

Compute and time

A web-scale pipeline runs on shared high-performance computing (HPC) clusters, where two different budgets govern what is possible. They are easy to conflate, so it is worth separating them now.

Compute time is the total machine work a run consumes, measured in **CPU hours** and **GPU hours**: one processor busy for one hour is one core-hour. It is what the cluster bills, and what a project's **compute budget** is fixed in.

Wall-clock time is the elapsed calendar time from start to finish. It is set by the longest chain of steps that must run in order, and by how much of the work can run in parallel. The same ten thousand core-hours can finish in an afternoon on enough machines or stretch over months on few.

The two come apart constantly: a stage can be cheap in compute yet slow on the clock, or the reverse. A real project is handed both a fixed compute budget and a deadline, so the two have to be managed together. Part 2 follows a concrete run where this split decides the shape of the whole pipeline.

With the vocabulary in place, Part 1 follows raw text through the pipeline that turns it into training data.

PART 1

The pipeline

How raw text becomes training data. We follow two paths, curated and web, from their very different starting points to a single shared output.

[2] The phrase is Joseph M. Juran's; its data-quality formulation, defining quality from the data consumer's point of view, is R. Y. Wang and D. M. Strong [2].

By the end of Part 1, you will be able to:

- Define **curated data** by the editorial process behind it, and say why curated and web sources take different paths through the pipeline.
- Trace a single document from raw input to the shared **JSONL record**, naming what each step cleans and what metadata it records.
- Explain why a pipeline produces two things at once, the **cleaned text** and the **record of how it was made**, and why that record is what makes work at scale possible.

Not all text is equal

A model learns from whatever text it is shown, so the composition of the data (its sources, its domains, its quality) shapes how the model behaves [3], [4].

Raw text from the internet is uneven:

- clean prose sits next to navigation menus, cookie banners, and boilerplate
- the same page can appear thousands of times
- some text is spam or machine-generated filler

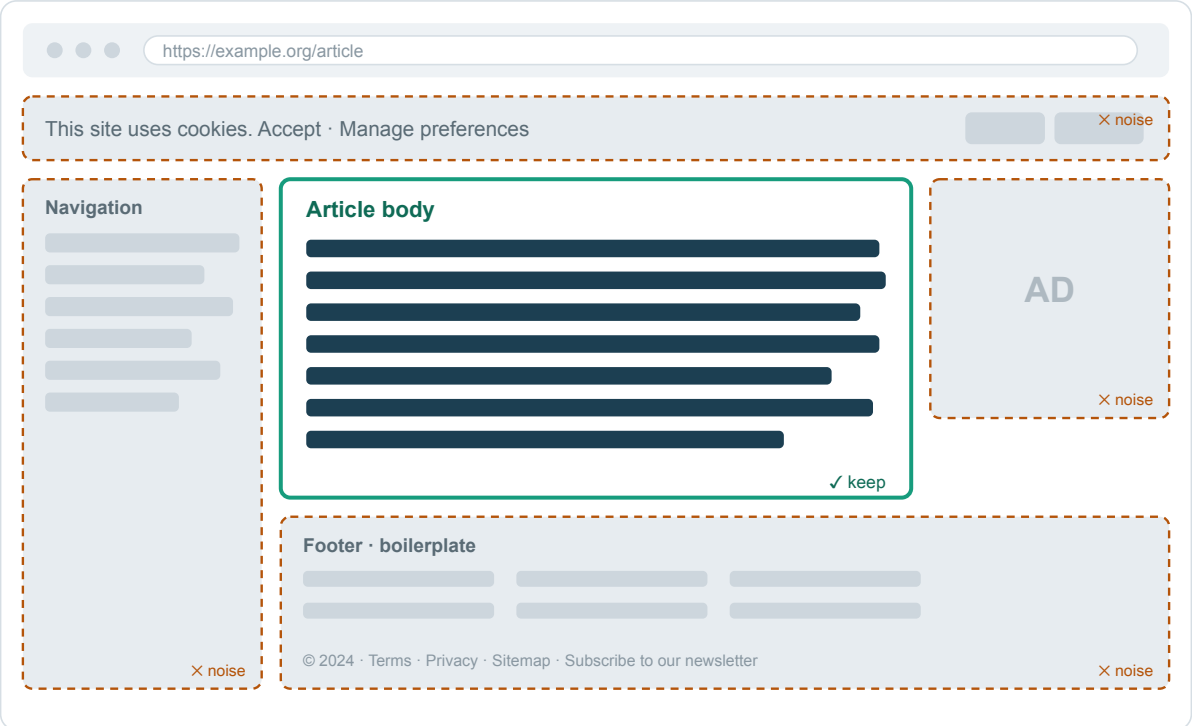


Figure 1: On a typical web page, the genuinely useful prose (the article body) is a small island in a sea of navigation, cookie banners, ads, and boilerplate. The pipeline's job is to keep the island and discard the rest.

More text is not automatically better text. The job of a data pipeline is to turn a large, messy pile of raw text into a smaller body of training data that is actually worth learning from.

The fork: curated vs web

That job starts by splitting the raw material by where it came from: two kinds of source need two kinds of handling.

Curated sources are collections that already passed through a human curation process before we ever saw them: a book has an editor, an encyclopedia has reviewers, parliamentary proceedings have an official transcriber. That editorial care, someone whose job was to make the text correct and coherent,

is what makes them relatively clean, relatively small, and high value per token. They take the *light path*, precisely because most of the quality work was already done upstream.

Web data comes from large crawls of the open internet. It is enormous and cheap to obtain, but mostly low value. It takes the *heavy path*: it needs aggressive cleaning before it is usable [4].

The next two sections follow each path from its raw input to the shared JSON record they both produce.

Curated pipeline

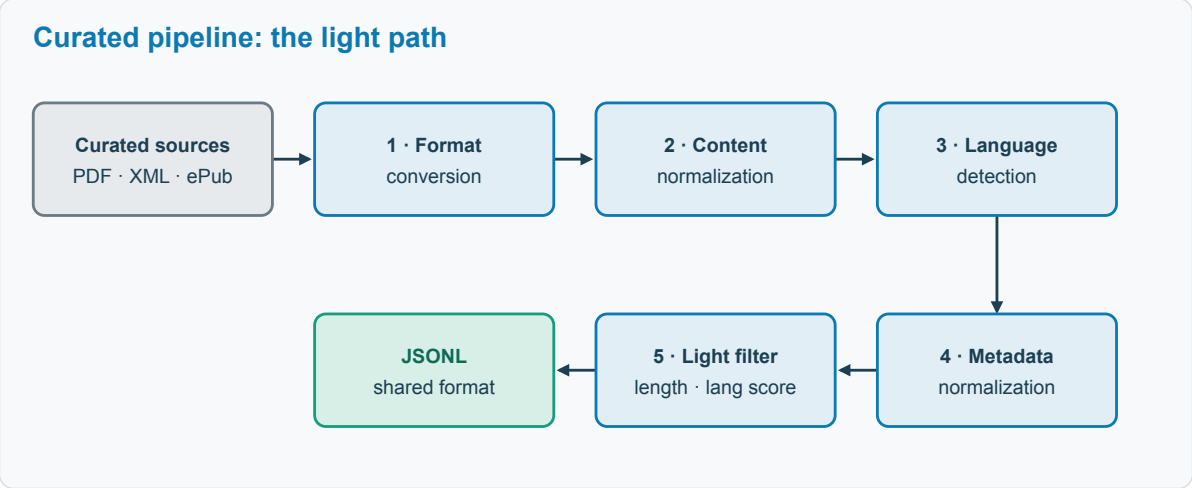


Figure 2: The curated path, following Figure 1 of N. Brandizzi *et al.* [5]: convert each source to plain text, normalize its content, detect its language, normalize its metadata, then apply light filtering. Both paths end in the same shared JSONL record.

The curated path is short because the inputs are already orderly.

Format conversion and content normalization

Curated sources arrive in many formats: PDF, XML, HTML, ePub. The first step is **format conversion**: extract the plain text[3] into one common representation, UTF-8 text with the source's markup and layout discarded. The second is **content normalization**: make the text valid, canonical UTF-8, standardize whitespace and line breaks, and remove stray control characters and conversion artifacts.

This is small and unglamorous work, but it removes a whole class of downstream errors before they can spread.

```
import unicodedata
import re

raw = "Café\u00a0\u00a0World\t\ttext   here\r\n\r\n"

def normalize(text):
    text = unicodedata.normalize("NFC", text)   # valid, canonical Unicode
    text = re.sub(r"\s+", " ", text)            # one regex collapses every
                                                # whitespace run: spaces, tabs,
                                               # newlines, even non-breaking spaces
    return text.strip()                         # trim the edges

print(raw)                  # on screen the hidden characters look like spaces
```

[3]Extraction is lossy and format-specific: a multi-column PDF must be read in the right order rather than straight across, and an HTML or XML file must yield its article body without the surrounding navigation and boilerplate. Getting either wrong corrupts the text before any later step sees it.

```
print(repr(raw))            # the real string: non-breaking spaces, tabs, returns
print(repr(normalize(raw)))  # after normalization: ordinary single spaces
```

```
Café  World     text   here


'Café\xa0\xa0World\t\ttext   here\r\n\r\n'
'Café World text here'
```

Printed, `raw` already looks clean, yet `repr` exposes the non-breaking spaces, tabs, and carriage returns hiding inside it. The third line shows the same text after `normalize`, where every whitespace run has collapsed to a single ordinary space. Two strings can look identical on screen and still differ character for character, and that invisible mismatch is what later lets one copy of a document slip past deduplication, or makes a search for a phrase miss it. Normalization makes equal-looking text actually equal.

Language detection, metadata, and light filtering

The Teuken models from the introduction span all 24 official EU languages. A multilingual model is trained on a deliberately balanced mix of those languages, so every document first has to be sorted into the one it is written in, something a monolingual (say English-only) pipeline never has to worry about. That means labeling each document with its language and a **confidence score**. That label, with a few other attributes (a unique id, the source URL, the date), is consolidated by **metadata normalization**. Every source describes its documents differently, so this step maps them all onto one schema. After it, every later step reads the same fields no matter where a document came from.

Finally, a *light* filtering pass drops a document that is too short or whose language score is too low to trust. The pass is light because curated inputs are already fairly clean: the goal is to catch obvious problems and otherwise leave the corpus as it is.

The interesting part is where that confidence score comes from. Here is a tiny detector that produces one: the language whose common "stop words" appear most is the guess, and the share of the document those words cover is the confidence. A document we cannot place gets the label `xx`, the same marker the real pipeline uses for non-language content.

```
# A toy stop-word language detector. Real pipelines use character n-gram models
# (e.g. fastText); this shows where a language label and its score come from.

STOPWORDS = {
    "en": {"the", "a", "is", "are", "and", "of", "to", "in", "it", "on"},
    "fr": {"le", "la", "les", "un", "une", "et", "de", "est", "dans", "ici"},
    "de": {"der", "die", "das", "und", "ist", "ein", "eine", "zu", "in", "den"},
}

def detect_language(text):
    words = [w.strip(".,!?;:") for w in text.lower().split()]
    hits = {lang: sum(w in sw for w in words) for lang, sw in STOPWORDS.items()}
    best = max(hits, key=hits.get)
    score = hits[best] / len(words) if words else 0.0
    return (best, score) if score > 0 else ("xx", 0.0)

MIN_CHARS      = 20    # too short to be useful
MIN_LANG_SCORE = 0.20  # below this share of known words, we do not trust the label
```

```
def keep(text):
    lang, score = detect_language(text)
    return len(text) >= MIN_CHARS and score >= MIN_LANG_SCORE

docs = [
    "The model is trained on a lot of raw text.",      # clear English
    "Le chat dort.",                                   # clear French, but short
    "Le chat est sur le tapis dans la maison.",        # clear French
    "Questo testo non contiene parole molto comuni.",  # a language we did not list
]
```

Real pipelines usually threshold on token counts. We count characters here only to keep the example self-contained.

! Your turn

First, predict the output. For each document, what language and confidence does `detect_language` return, and does `keep()` keep it?

Then the harder question. `keep()` only sees length and a language score. Sketch a document that would clear *both* thresholds and still be worthless as training data. What does that tell you about what a filter like this can and cannot judge?

i Solution

The detector's verdict on each document, computed at render time:

```
for d in docs:
    lang, score = detect_language(d)
    print(f"{lang} {score:4.0%}  {'keep' if keep(d) else 'DROP'}  {d}")
```

```
en  50%  keep  The model is trained on a lot of raw text.
fr  33%  DROP  Le chat dort.
fr  56%  keep  Le chat est sur le tapis dans la maison.
xx   0%  DROP  Questo testo non contiene parole molto comuni.
```

Two survive. The English and French sentences are recognized confidently, since about half their words are common function words, so they clear `MIN_LANG_SCORE`. `"Le chat dort."` is dropped by `MIN_CHARS`: at thirteen characters it is too short, even though the detector is fairly sure it is French. The Italian sentence is long and perfectly good text, but our detector knows no Italian stop words, so it scores 0.0, is labeled `xx`, and is dropped. A low score does not mean bad text; it means the pipeline cannot vouch for the label, and data it cannot place by language has nowhere to go in a per-language corpus.

The harder question is the real point. A long, confidently-English block of spam clears both thresholds easily. Cruder cases are caught later by the web path's quality metrics and blocklists, but the deeper gap, between what a filter can measure and whether the text is any good, is the problem Part 3 takes up.

Our toy detector fails on Italian because it was never given Italian stop words, and that exposes two failure modes real pipelines hit as well. **Low-resource languages are hard to detect:** a detector

trained on little text for a language produces low-confidence labels, so genuine documents in that language score poorly and get dropped, exactly as our Italian sentence did. **Similar languages are hard to distinguish:** closely related languages share much of their vocabulary, so a confident label can still be the wrong one (a stop-word detector would struggle to tell Spanish from Portuguese, or Danish from Norwegian). The label and its score are only ever as good as the model behind them.

> **i** Useful tools
>
> Production pipelines do not hand-roll these steps. Language detection usually runs fastText's `lid.176` model instead of stop-word counts, and Unicode normalization is left to a library like `ftfy`, which repairs the mojibake and encoding glitches that a plain whitespace pass misses.

Web pipeline

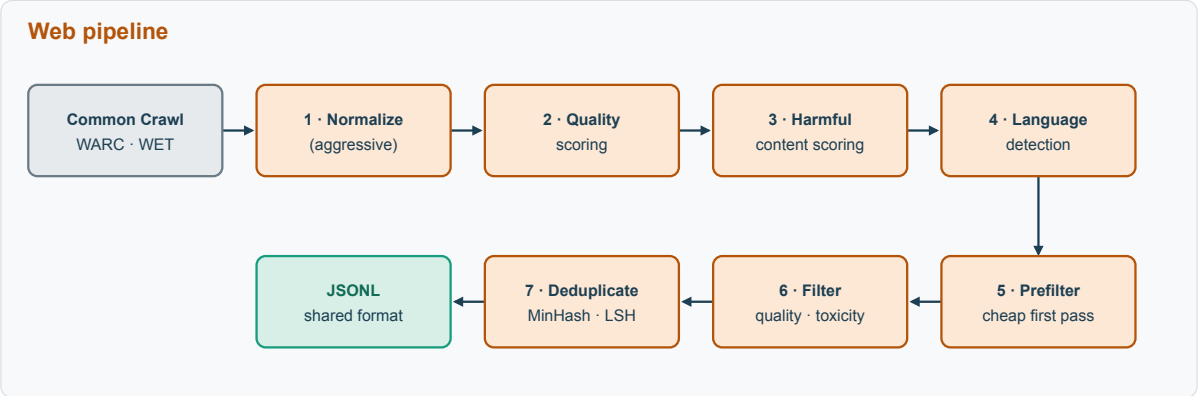


Figure 3: The web path, following Figure 2 of N. Brandizzi *et al.* [5]: from Common Crawl, normalize aggressively, score for quality and harmful content, detect language, then prefilter, filter, and deduplicate before reaching the same shared JSONL record.

The web path is longer because the input is raw and mostly unusable. The end goal is the same, but it takes many more steps.

Acquisition

The common web source is **Common Crawl**, a public crawl of the open web that is republished regularly. Each release is called a *dump*.

A dump can be read in two forms:

- a **WARC** file holds the full raw HTTP response, including the HTML markup
- a **WET** file holds only the plain text already extracted from that HTML

WET is cheaper to process but lower fidelity. Extracting text yourself from WARC keeps more of what matters and gives cleaner results, but it costs more on two fronts: more compute to parse the raw HTML, and more engineering work, because you build the extraction and its filters yourself and have to test that they do what you intend. The trade-off is quality against both compute and effort.

Even a single dump is large, and the full set runs to hundreds of terabytes.

The processing core

After acquisition, each document passes through a sequence of scoring and detection steps. The point here is what gets *measured*; the actual thresholds come next:

- **normalize**, as in the curated path but more aggressively
- **quality scoring**: a few cheap metrics that estimate how document-like the text is
- **harmful-content scoring**: a separate score for how likely the text is to be toxic or unsafe
- **language detection**: a language label and a confidence score

- **prefilter**: a cheap first pass that discards the clearly unusable

These steps work in tandem: the scoring steps attach a score to each document, and the filter later turns those scores into keep-or-drop decisions. Each is picked up in detail below: the quality metrics and the prefilter under *Filtering*, the harmful-content score under *Harmful content*, and language detection exactly as on the curated path.

Filtering

Filtering turns each measurement from the processing core into a keep-or-drop decision, cheapest checks first so the expensive ones only ever see documents that already survived. Most quality signals are just *metrics*: a cheap count of something about a document. A filter then compares that metric to a threshold and keeps or drops the document. A few simple signals catch a surprising amount of junk:

- **average words per line**: menus, link lists, and tables are mostly very short lines
- **share of symbols and digits**: code, logs, and gibberish are heavy on non-letters
- **document length**: too little text carries too little signal
- **repeated-line ratio**: navigation and boilerplate repeat the same lines

A prefilter is just a few of these metrics behind thresholds. Here is one made concrete, on three example documents:

```
def words_per_line(text):
    lines = [ln for ln in text.splitlines() if ln.strip()]
    return len(text.split()) / len(lines) if lines else 0.0

def symbol_share(text):
    symbols = sum(1 for c in text if not (c.isalpha() or c.isspace()))
    return symbols / len(text) if text else 1.0

def looks_like_prose(text):
    return words_per_line(text) >= 4 and symbol_share(text) <= 0.30

examples = {
    "article":  "Warmer winters are projected for the region.\n"
                "Researchers compared decades of temperature records.",
    "nav menu": "Home\nAbout\nProducts\nContact\nLogin\nHelp",
    "log dump": "[03:11:02] ERR 0x8F retries=3 code={id:9} >>> 11010 0xFF",
}

for name, text in examples.items():
    verdict = "keep" if looks_like_prose(text) else "DROP"
    print(f"{name:9} words/line={words_per_line(text):4.1f}  "
          f"symbols={symbol_share(text):3.0%}  ->  {verdict}")
```

```
article   words/line= 6.5  symbols= 2%  ->  keep
nav menu  words/line= 1.0  symbols= 0%  ->  DROP
log dump  words/line= 8.0  symbols=50%  ->  DROP
```

The article keeps; the navigation block fails on words per line, and the log dump fails on symbol share. What to notice is *how* these metrics judge: `symbol_share` reads every character, but it does not understand a single one. It measures the document's surface statistics rather than what the text means or whether it is any good. That is exactly what makes the check cheap enough to run on hundreds of terabytes, and it is also its blind spot: a fluent, grammatical paragraph of marketing fluff or confident

nonsense scores perfectly. Judging *meaning* is a different and much harder problem, and it is the subject of Part 3.

Harmful content

Quality is not a single axis, though. The metrics above all ask *is this document-like?* A separate question is *is this safe?* Training data carries the **toxicity and harmful content** of its sources straight into the model's behavior, so pipelines add a distinct **harmful-content** signal alongside the quality ones, dropping the worst material however clean the document's shape looks [6].

This check runs *before* language detection (Figure 3) because it needs no language label: a **domain blocklist** drops whole unsafe sites before they are fetched, and multilingual **word lists** catch documents heavy with slurs and profanity across languages.

Filtering is where most of the raw volume disappears.

> **i** Useful tools
>
> Both levels start from public resources rather than a trained model: multilingual profanity lists such as LDNOOBW for the word-level check, and a domain blocklist such as the UT1 blocklist for the site-level one.

Deduplication

The web is extremely repetitive. The same article, quote, license notice, or template appears countless times. Training on duplicates wastes compute and can push the model toward memorizing instead of generalizing, so duplicates have to go [7].

Exact duplicates are easy to spot. The hard case is **near-duplicates**: pages that differ only in a date, a header, or an ad. The shared idea is *similarity*: measure how much two documents overlap, and drop one when the overlap is too high. Here is that idea in miniature, scoring documents by the overlap of their word sets (the Jaccard similarity):

```
corpus = [
    "The quick brown fox jumps over the lazy dog.",
    "The quick brown fox jumps over the lazy dog.",         # exact duplicate
    "The quick brown fox jumps over the lazy dog. (2024)",  # near-duplicate: a date
    "An entirely different sentence about another topic.",
]

def word_set(text):
    return set(text.lower().replace(".", "").split())

def jaccard(a, b):
    return len(a & b) / len(a | b) if (a | b) else 0.0

sets = [word_set(doc) for doc in corpus]
n = len(corpus)

# Score every pair into a similarity matrix. It is symmetric and its diagonal is
# always 1.0, so the real information lives in the upper triangle (j > i).
sim = [[jaccard(sets[i], sets[j]) for j in range(n)] for i in range(n)]
print("similarity matrix:")
for row in sim:
    print("  " + "  ".join(f"{value:.2f}" for value in row))
```

```python
# Walk the upper triangle once. Drop document j when it repeats an earlier,
# still-kept document i.
THRESHOLD = 0.5
dropped = set()
for i in range(n):
    if i in dropped:
        continue
    for j in range(i + 1, n):
        if sim[i][j] >= THRESHOLD:
            dropped.add(j)

print("\nverdicts:")
for i, doc in enumerate(corpus):
    print(f"  doc {i}  {'DROP' if i in dropped else 'keep'}  {doc}")
```

```
similarity matrix:
  1.00  1.00  0.89  0.00
  1.00  1.00  0.89  0.00
  0.89  0.89  1.00  0.00
  0.00  0.00  0.00  1.00

verdicts:
  doc 0  keep  The quick brown fox jumps over the lazy dog.
  doc 1  DROP  The quick brown fox jumps over the lazy dog.
  doc 2  DROP  The quick brown fox jumps over the lazy dog. (2024)
  doc 3  keep  An entirely different sentence about another topic.
```

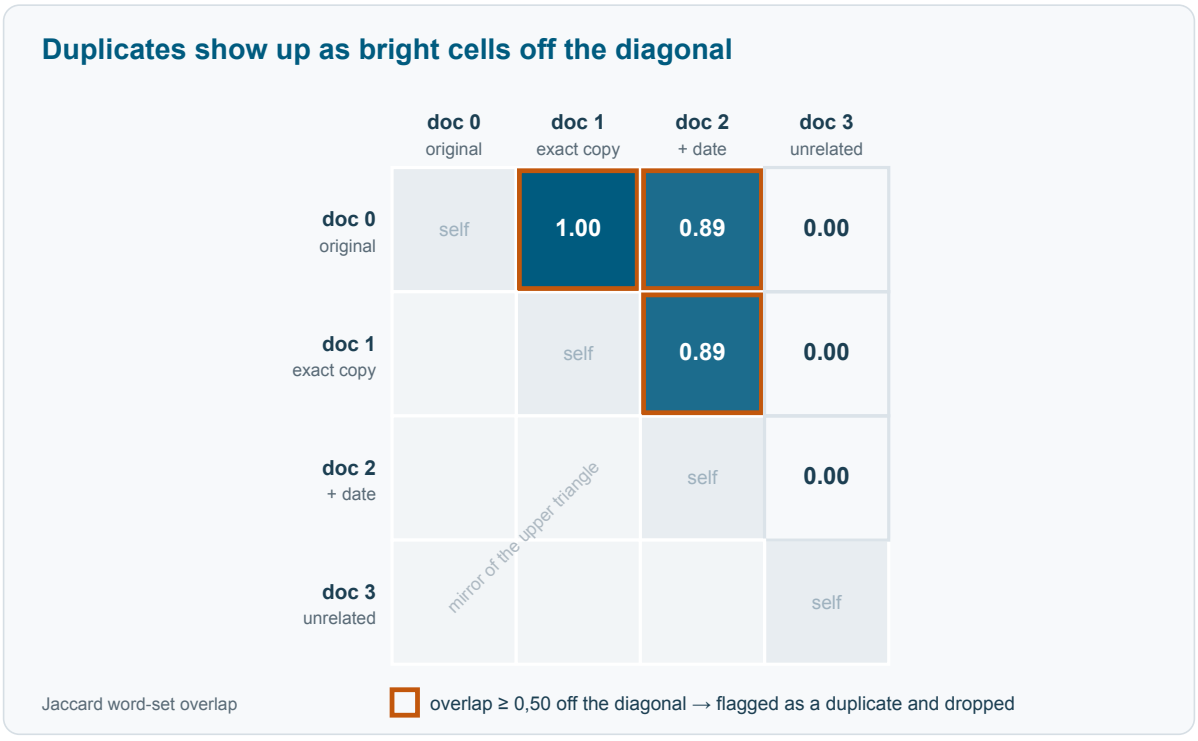


Figure 4: The same similarity matrix as a heatmap. The bright off-diagonal cells are the duplicate pairs: document 0 against its exact copy (1.00) and its date-stamped near-copy (0.89).

In the upper triangle, document 1 scores 1.00 against document 0 (an exact copy) and document 2 scores high (the same text plus a date), so both are dropped as repeats of document 0; the unrelated document overlaps little with everything and stays. Comparing every pair of documents this way is far too slow at web scale, which is why OpenGPT-X uses **MinHash** with **LSH** (locality-sensitive hashing) [8] to estimate the same overlap cheaply, without comparing all pairs. The compute cost is still large, and we return to it in Part 2.

> **i** Useful tools
>
> The near-duplicate method here has mature open implementations: `datasketch` for MinHash and LSH in Python, `text-dedup`, which packages the full near-deduplication workflow, and `deduplicate-text-datasets`, the suffix-array code released with K. Lee *et al.* [7].

Both paths converge

Whatever the source, every surviving document ends in the same shape: one JSON object per line, a format called **JSONL**. Each record carries the cleaned text plus the metadata gathered along the way: source, language, scores, and identifiers.

```
{
  "id": "web-2024-30-000123",
  "text": "The cleaned document text, ready for training.",
  "source": "common_crawl",
  "language": "de",
  "language_score": 0.98,
  "quality_score": 0.71,
  "url": "https://example.org/article",
  "timestamp": "2024-07-15"
}
```

On disk each record sits on a single line; it is pretty-printed here only for readability. Fields are illustrative. This shared format is what lets curated and web data be combined downstream: mixing the sources in chosen proportions, versioning the corpus, and tokenizing it for training are all later steps, beyond this document, but each becomes possible only because every source now shares one shape.

Keeping the *scores*, not just the keep-or-drop decision, has a second payoff that matters at scale. The recorded scores are the signals, and a threshold like `MIN_LANG_SCORE = 0.20` is the **filter decision** laid over them. A threshold is a choice, and choices get revisited. The pipeline computes the signals once; the filter decisions stay adjustable afterward.

> **!** Your turn
>
> Picture a corpus built *without* keeping any of this metadata: just the final text, no source, no scores, no language label. Months later, the model trained on it turns out weak in one language. With only the raw text in hand, how would you even begin to find out why?
>
> Now look back at the JSONL record above. Which fields, had they been kept, would let you answer that, and what would each one tell you?

One way to reason about it

- **`language` and `language_score`:** was the language detected at all, and how confident was the detector? A low score points at misclassified or mixed-language text.
- **`quality_score`:** were the surviving documents actually good, or did a loose threshold let filler through?
- **`source`:** is the weak language coming mostly from one source that behaved badly?
- **`id` and `timestamp`:** can you locate and re-pull the exact documents to inspect or re-run them?

The point: without the record, none of these questions are answerable. The metadata is what makes a fault findable.

Recap

Two very different inputs, curated and web, now share one shape: clean text paired with the metadata that records where it came from and how it was scored. That pairing is the real result of Part 1, and it is two outputs at once, both of which have to hold: the **quality** of the data, and the **traceability** of how it was produced.

So far we have treated each step as a matter of logic: what to keep, what to drop, what to record. Part 2 changes the question. At the true size of these corpora, every one of those steps is also a compute decision, and the traceability we just built is what makes the whole effort survivable.

PART 2

Working at scale

When you treat the pipeline as pure logic, every step has a clean answer: keep this, drop that, record the rest. Put it at the true size of a web corpus and a second constraint appears behind every one of those answers, a *price* measured in machine time. This part is about what changes once compute is the thing in short supply, because some of the answers from Part 1 stop being affordable and others stop being possible.

By the end of Part 2, you will be able to:

- Define **compute time** and **wall-clock time** as two separate budgets.
- Reframe a **97% removal rate** as the expected outcome at web scale.
- Explain why **deduplication** is the biggest compute cost yet not the slowest stage, and why that fixes its place in the pipeline.
- Identify **ordering, storage, and lineage** as the load-bearing engineering decisions that emerge at this scale.

Scale changes the problem

The web side of Teuken's data started from **60 Common Crawl dumps**, spanning late 2014 to early 2023. Together they held about **173 billion documents**, roughly **703 terabytes** of raw text, around 2,7 billion documents per dump. Numbers this large change what the steps from Part 1 *mean* in practice.

Take filtering. On average, **96,94% of web documents are removed during filtering** [5]. The introduction set out why a number like that is a sign of success rather than failure: at web scale the input is *known* to be mostly navigation, boilerplate, and spam, so aggressive removal is the job working as intended. What the headline figure hides is how much still survives. Roughly 3% of 173 billion documents is on the order of **five billion documents** of far cleaner text, a large corpus in its own right.

Numbers this size are hard to feel, so start from a landmark and work outward. The founding of Rome was about **2.800 years ago**, already deep history. Yet reading only the five billion documents that *survive* filtering, at a normal pace and never stopping, would take some **35.000 years**, more than ten times as far back as Rome, as old as the first cave paintings. And the full raw collection of 173 billion documents would take on the order of **1,2 million years**, about four times longer than our species has existed.

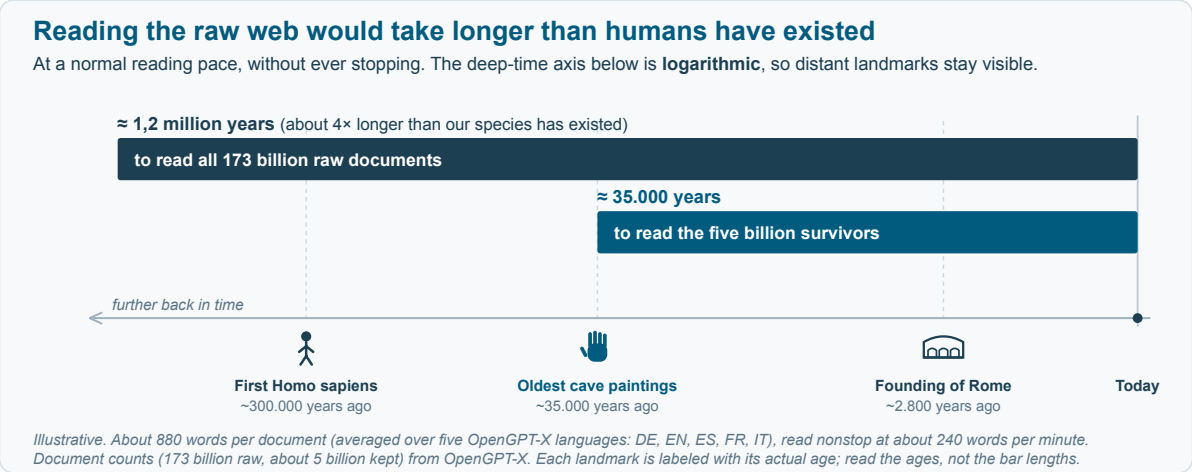


Figure 5: How long it would take a person to read the raw web, and the five billion documents that survive filtering. Read each bar by the age it reaches, not by its length.

A web corpus already holds far more text than any model needs, so the binding constraint becomes machine time. From here on the work is rationed by compute, and that budget is spent very unevenly across the steps.

Every decision is a compute decision

Once a step has a price, ordering it is an optimization. The pipeline runs its **cheapest checks first** so the expensive ones only ever see documents that already survived. A prefilter that costs almost nothing per document removes the obvious junk before any costly scoring or comparison runs, which means the expensive stages process a far smaller pile.

Scale also turns two more concerns into load-bearing engineering decisions:

- **Lineage and versioning.** Each stage writes its output in the same JSONL format and saves intermediate results, so any version can be compared with the one before it and a fault can be traced to the stage that introduced it. This is the traceability from Part 1.
- **Storage and movement.** Those intermediate results are copied to remote storage to guard against loss, which costs disk; and moving hundreds of terabytes between a processing cluster and a training cluster is slow and expensive enough that keeping the data where the model will train is itself a design choice.

None of this is visible in the final corpus, yet it is what keeps a multi-month pipeline from becoming unrepeatable.

Take the failure from Part 1′s metadata exercise, the corpus that turns out weak in one language (Part 1). If the pipeline saved its outputs after language detection and after later filtering, the two versions can be compared and the loss traced to the stage that introduced it. If those intermediate snapshots were discarded to save storage, diagnosing the failure turns into rerunning months of work.

i Further reading

Poolside's model factory describes an industrial version of exactly this idea: treating data processing as a versioned, monitored production line rather than a one-off script.

Deduplication is the expensive step

The clearest example of a logical step that is really a compute decision is deduplication. Of all the CPU time spent processing the web data, deduplication consumed about **80,8%**, with filtering at 16,7% and format conversion at 2,5% [5]. Put in human terms, the three stages together come to about **273.952 CPU hours**, close to **31 years** of work for a single CPU core. Deduplication is around **25 of those years** on its own. A cluster runs it in months by splitting the work across many cores, but the raw cost is the size of three decades on one machine.

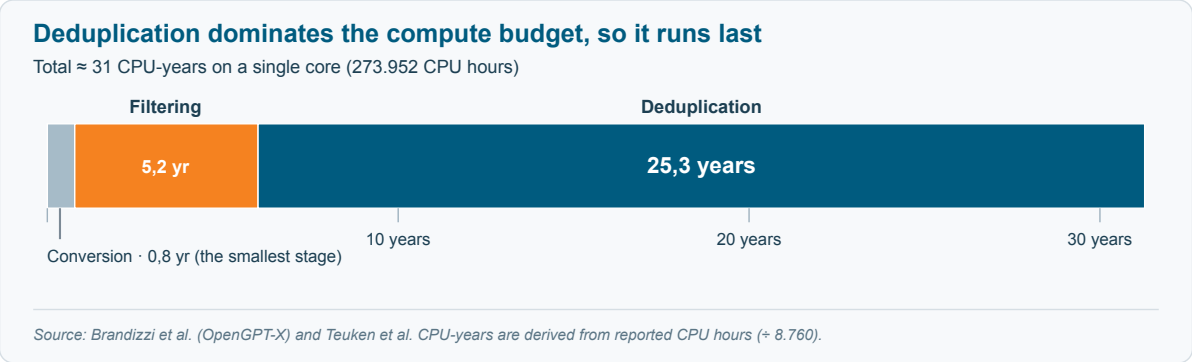


Figure 6: How the web pipeline's compute splits across its three stages.

That single number sets the pipeline order. Because comparing documents for similarity is so costly, you run it **last**, on the small fraction that survived everything else, never on the full raw pile. Order the stages the other way and you would pay the most expensive step on the 97% you were about to discard.[4]

! Your turn

Deduplication is the most expensive stage and runs last; filtering removes about 97% of documents and runs earlier. Suppose an engineer, wanting "the cleanest data first," reorders the pipeline to deduplicate *before* filtering. The output documents are identical either way. So what exactly does the reordering cost, and roughly by how much?

[4]The same cost also forces a choice of *scope*: comparing every document against every other across all 60 dumps (a *global* deduplication) was impractical, since for a high-resource language like English the combined set was simply too large to process and brought no measurable benefit to the trained model. So deduplication was done *locally*, per dump and per language, which still removed about 30% of the data while staying inside the compute budget, the same logical goal with a different answer once the constraint is compute rather than correctness.

i One way to reason about it

The factor that drives the cost is the **number of documents** deduplication has to compare: more documents means more comparisons, so the cost rises with the size of the input pile. Since filtering removes about 97% of documents, the "dedup first" order makes the most expensive stage process roughly 30 times as many documents for the same final corpus. Put in the same single-core terms as above: the ~25 core-years deduplication already costs would balloon to **on the order of 750 core-years** to produce an identical output. That is the whole reason the cheap, decisive cuts come first and the costly comparison comes last.

Compute time versus wall-clock time

Total CPU time measures the work; how long the calendar takes is a separate question. Every stage already spreads across thousands of cores, so they all parallelize equally well. What separates them, and sets the calendar, is **throughput**: how much data each stage clears per day, which depends on how much work a single document costs it.

Deduplication makes the point. It is the largest compute cost of all, four-fifths of the budget from the previous section, yet it moves through the data quickly: the cluster clears about 20 terabytes of raw data a day. The reason is the *kind* of work it does. Deduplication compares documents with a simple whitespace tokenizer and numeric similarity algorithms, cheap operations per document, so even though there are an enormous number of comparisons, each one is fast.

Computing the quality signals is the opposite case. Every document is read in full: content analysis and tokenization, the heavy lifting of the pipeline, run on each one, so the stage clears only about **6 terabytes a day**. Downloading the raw data is slow for a different reason: its speed is set by network bandwidth from the source, not by how many cores you add, and it manages a similar **6 terabytes a day**. These two stages, three to four times slower than the rest, set the pace, because the whole pipeline can finish no faster than its slowest stage.

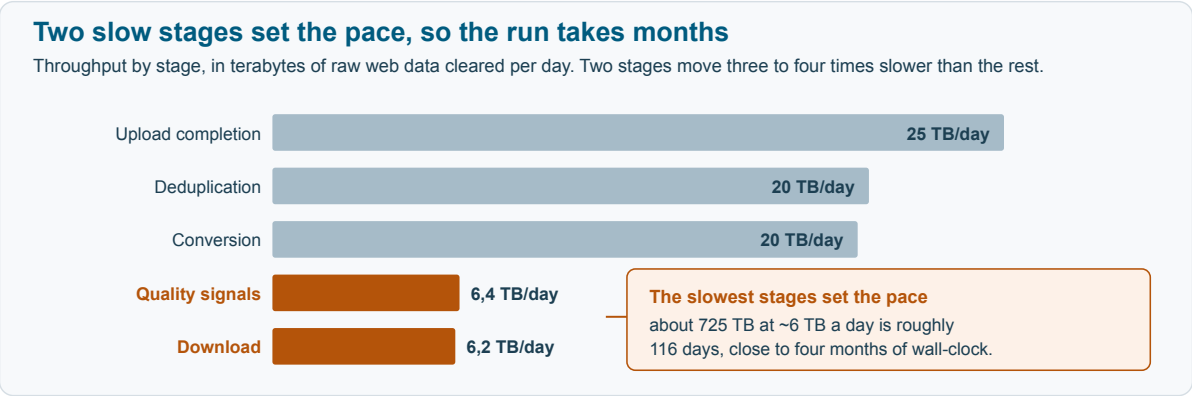


Figure 7: Per-stage throughput, and the two slow stages that stretch the run to months. Bars show raw data cleared per stage per day, from the OpenGPT-X processing run of about 725 TB of raw web text, broader than the 60-dump set used for the final corpus.

This is the gap between *compute* and *time*. Pushing roughly **725 terabytes** of raw web text through the pipeline, held to about 6 terabytes a day at the two slow stages, takes on the order of **116 days**, close to four months, even on a large cluster.

And wall-clock time, not total compute, is the budget that actually binds. It is tied to the project's lifetime: a project is allotted a compute window, and the corpus has to be ready inside it. Four months of wall time inside a one-year project is a very different proposition from the same corpus needing two years. Total core-years is what you pay for; wall-clock time is what you live by.

Key takeaway

Total compute time and wall-clock time are two separate budgets, and a real project is handed both at once: a fixed compute allocation and a deadline. Wall-clock time, tied to the project's lifetime, is the one that ultimately binds, so the job is to fit the corpus inside the deadline without spending more than the compute you were given.

Recap

Scale does not add new steps so much as it re-prices the ones we already have. Once every decision carries a cost in machine time, the question behind each step shifts from *what is correct?* to *what is affordable?* That shift is the lesson of this part: at web scale, preparing data becomes an exercise in economics as much as logic. The order of the pipeline, the reach of a step like deduplication, even whether a step is possible at all, are settled by the compute budget rather than by correctness alone.

Both parts so far have rested on one assumption: that cheap, surface-level signals can separate good text from bad. Part 3 puts that assumption to the test.

PART 3

The quality question

Part 1 built a pipeline that asks one question of every document: *is this well-formed?* The cheap surface checks answered it by counting characters and lines, and Part 2 showed that they scale to hundreds of terabytes precisely because they are so cheap. This part asks the question those checks cannot reach: of the documents that are well-formed, which ones are actually worth learning from? That is the quality question, and it is where the easy answers end.

By the end of Part 3, you will be able to:

- Distinguish a document's **form** (is it well-formed?) from its **value** (is it worth learning from?).
- Explain why **surface heuristics**, for all their speed and transparency, cannot see content value.
- Compare an **LLM judge** against a surface heuristic in real terms: what it gains in judgment, and what it costs in money and hardware.

Surface heuristics: cheap, transparent, blind

The surface metrics from Part 1, the cheap quality signals it filtered on, are **surface heuristics**: fixed rules that measure a document's *shape*, how its characters and lines are arranged, the same way for every document. They cost almost nothing per document, which is what lets them run across web scale. They are transparent: a rule like "drop documents with fewer than four words per line" can be read, audited, and reasoned about. And they are deterministic, so the same document always gets the same verdict.

What they cannot do is see *value*. A document can be perfectly well-formed and still be worthless to train on, and a heuristic, counting only surface features, gives it the same verdict as a genuinely useful one. And value takes more than one form: a document can be judged on how densely it packs information, on how well it is written, on how accurate it is. Here are two documents checked by the **very same prefilter from Part 1**, `looks_like_prose` remains unchanged:

```
# words_per_line(), symbol_share(), and looks_like_prose() are the Part 1 prefilter.
```

```
documents = {
    "explainer": "Photosynthesis converts sunlight into chemical energy.\n"
                 "Plants combine water and carbon dioxide to build sugars.",
    "spam":      "Best cheap flights, book cheap flights online today.\n"
                 "Cheap flights deals, find cheap flights and save now.",
}

for name, text in documents.items():
    verdict = "clean" if looks_like_prose(text) else "DROP"
    print(f"{name:9} words/line={words_per_line(text):4.1f}  "
          f"symbols={symbol_share(text):3.0%}  ->  {verdict}")
```

```
explainer words/line= 7.5  symbols= 2%  ->  clean
spam      words/line= 8.5  symbols= 4%  ->  clean
```

Both pass. The same prefilter that dropped Part 1's nav menu and log dump keeps these two, yet only one is worth training on. The prefilter cannot tell them apart, because it counts the shape of the text and never reads what the words mean.

Form and value as two axes

Picture every document placed on a grid. One axis is *form*, how well-formed the document is; the other is *value*, how much it is worth learning from. A surface heuristic reads only the horizontal axis, so any filter built on it can draw just one kind of line: a vertical cut that keeps the well-formed half and drops the rest, whatever those documents are worth. On the two diagonal cases that cut is right. A well-formed, valuable article is kept, and a malformed, worthless log dump is dropped. The other two corners are where it fails. A well-formed but worthless page, the spam from a moment ago, is kept; a substantive page that happens to look messy, say dense code or terse notes, is dropped. Those two corners are the price of judging form alone.

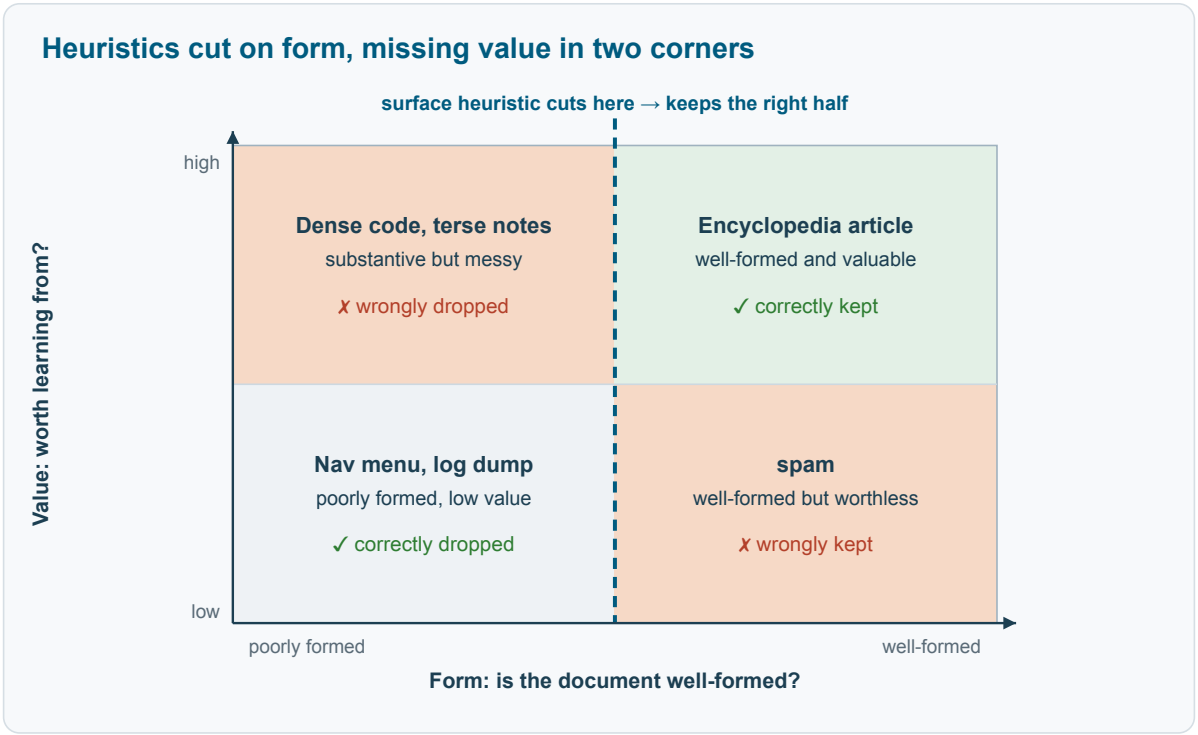


Figure 8: Surface heuristics judge a document's form, so they cut along one axis only. The two shaded corners, well-formed but worthless and substantive but messy, are where a form-only heuristic gets it wrong.

Heuristics do the job they are built for, cheaply and at scale, and they remain the first line of every pipeline. The point here is narrower: form is all they can measure, so value needs a different instrument.

> ! Your turn
>
> The heuristic above scored the spam and the explainer the same. Suppose you are allowed to add as many surface rules as you like: line length, punctuation balance, repeated-phrase ratio, vocabulary size, anything that counts characters or words without understanding them. Sketch a worthless document that still passes every such rule. What property of a document are all of these rules structurally unable to see?

> i One way to reason about it
>
> Such a document is easy to write: vary the vocabulary, balance the punctuation, keep every line a sensible length, and it passes every count while saying nothing worth learning. Surface rules measure how text is *arranged*, never what it *means*, so **meaning** is the one property they cannot reach. Reading meaning is what a language model does, which is where the next section goes.

Asking a model to judge

If the missing instrument has to read meaning, the obvious candidate is a language model: hand it the full document and ask it to score the value the heuristics could not see. Scoring value means choosing *which* kind of value to measure. The one put to work at web scale is **educational value**: how much genuine, learnable information a document carries, rated on a small scale such as 0 to 5 [4]. Because the model actually reads the text, it can tell the explainer from the spam in a way no counting rule can.

> i Further reading
>
> HuggingFace's FineWeb blog post walks through building exactly such a judge: an LLM scores documents for educational value, and those scores train the FineWeb-Edu quality classifier that filters a web-scale corpus.

The difficulty is that this power is expensive. A surface heuristic is a few arithmetic operations per document; an LLM judgment is a full forward pass through billions of parameters, far more work each time. Multiply that by the roughly five billion surviving documents, and compute becomes the real obstacle.[5]

The cost in euros

Of those two prices, the compute one can be made concrete. When you build a pipeline like this, the budget decides what you can actually run, as much as the quality you are aiming for does. The budget depends on two things: how much work there is, and what hardware it runs on[6].

CPU work and GPU work

The two kinds of processor are good at different things. A **GPU** packs thousands of small cores that all run the same arithmetic at once, which makes it fast at the dense matrix multiplication inside a

[5] A second price is cross-lingual certainty: Teuken's data spans all 24 EU languages, around 60% of it outside English, and whether one judge can rate quality evenly across every one, including low-resource languages it barely saw in training, is an open question. We follow the compute price here, since that is what the next step answers.

[6] The numbers below are order-of-magnitude illustrations at mid-2026 cloud prices; prices move, so the ratios are the durable part rather than the exact euro amounts.

neural network and slow at much else.[7] A **CPU** has a few powerful, flexible cores built for sequential logic: branching, comparing, and shuffling data around. Most of a data pipeline is that second kind of work, converting file formats, hashing and comparing documents, reading and writing bytes, none of it matrix arithmetic. So it runs on CPUs, where an hour of compute is cheap.

Recall the Part 2 budget. Conversion, filtering, and deduplication together took about **273.952 CPU hours**. At a representative cloud price of four to five cents per CPU hour,[8] that whole pipeline costs on the order of **12.000 €** to run once. For hundreds of terabytes of processed text, that is remarkably little.

Now imagine running the same work on GPUs, on the intuition that GPUs are faster. A GPU hour costs roughly **seventy times** a CPU hour,[9] and the work gains nothing from one: hashing and string comparison cannot use the matrix hardware you would be paying for. The same pipeline would cost on the order of **800.000 €**, and finish no sooner.

> Key takeaway
>
> Pay for a GPU only where the work is the dense, parallel arithmetic it is built for. Everything else belongs on a CPU, where it runs just as fast for a fraction of the price.

The judge at web scale

That rule has one exception in this pipeline, and it is the LLM judge. Reading a document for meaning *is* the dense matrix arithmetic a GPU is built for, so here the expensive hardware earns its price. The bill now has two multipliers working together: a GPU hour is expensive, and there are billions of documents to score.

A heavy judge, a model in the tens of billions of parameters, scores on the order of **ten thousand documents per GPU hour**.[10] At about 3 € per GPU hour that is roughly **0,0003 € per document**, which sounds negligible until you count the documents. Filtering leaves on the order of **five billion**, so judging all of them once comes to about **1,5 million €** and around **half a million GPU hours**, more than fifty years of nonstop work on a single card.

The shape is the one Part 2 made concrete: a cost that is tiny for one document grows enormous once it is paid billions of times over. Plotted against the number of documents, the effect is plain. Judging a small sample is inexpensive; judging the full corpus of survivors costs millions of euros:

[7] A third class of chip, the **tensor processing unit** (TPU) and a growing field of similar AI accelerators, targets the same dense matrix math as the GPU; for a data pipeline the decision that matters is still CPU versus GPU. Further reading: Google's introduction to TPUs.

[8] Representative on-demand Linux CPU price, about 0,045 € per vCPU-hour, roughly 0,034 € to 0,055 € across AWS, GCP, and Azure general-purpose instances, mid-2026. Cloud prices change; the figure is illustrative.

[9] An NVIDIA H100 runs about 3 € per GPU-hour on specialist clouds and 7 € to 12 € on the large providers, mid-2026. Against about 0,045 € per CPU-hour that is roughly 70 to 250 times more; we use the low end, which is the most favorable to the GPU. Prices change.

[10] Order-of-magnitude estimate: a roughly 70-billion-parameter judge in fp8 on an H100, batched, scoring documents of about 690 tokens each, processes on the order of 10.000 documents per GPU-hour, derived from public NVIDIA inference benchmarks. Actual throughput varies with model, hardware, and batching.

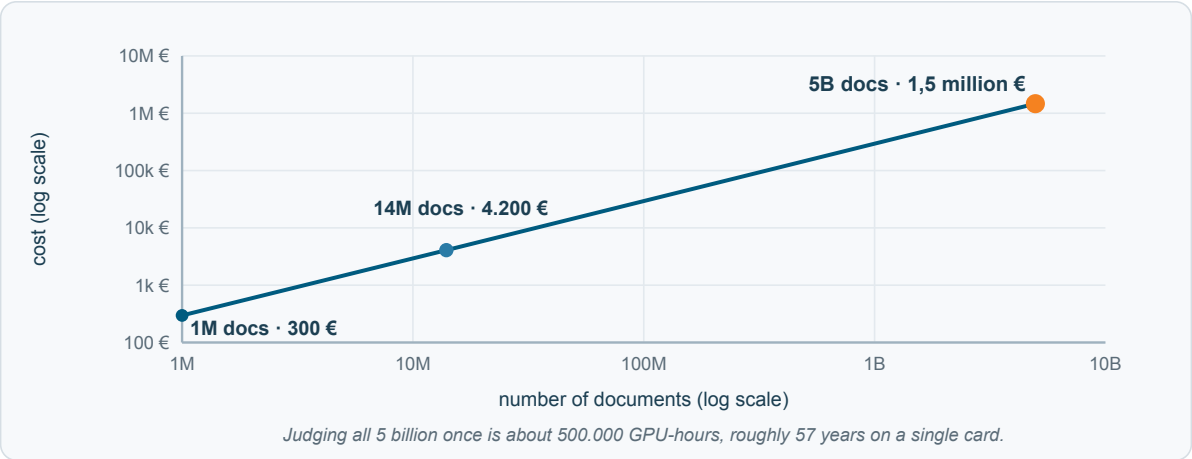


Figure 9: The cost of judging documents with a heavy LLM, as the number of documents grows. Affordable on a sample, a major expense across the web.

So the obstacle is the economics. A heavy judge reads meaning more reliably than any counting rule, yet running one across the whole web costs millions of euros and years of GPU time. That turns the next step into a sharp question: can a pipeline keep the judgment of a language model while paying something close to the price of a surface heuristic?

Recap

Part 3 separated two questions that Part 1's surface heuristics had treated as one. *Form* asks whether a document is well-formed; *value* asks whether it is worth learning from. Heuristics settle form cheaply and at any scale, but they measure only the shape of the text, so value stays invisible to them. A language model can read the text and judge its value, but doing that across billions of documents is slow and expensive.

That is the open problem this part leaves on the table: how to get the judgment quality of a language model at the running cost of a surface heuristic. Part 4 is the answer this material follows, and it solves the problem with a move in the spirit of Part 2: trade a large one-time compute cost for a cheap one that runs at scale.

> **i** Useful tools
>
> Open-source frameworks package these stages so you do not build them from scratch: DataTrove (the pipeline behind FineWeb) and NVIDIA's NeMo Curator both handle filtering, deduplication, and large-scale processing. For the LLM-judge idea specifically, propella-1 annotates documents across many quality properties at once rather than a single score, and releases the resulting annotations as a dataset.

PART 4

Cheap proxies for good text

Part 3 ended on a sharp question: can a pipeline keep the judgment of a language model while paying something close to the price of a surface heuristic? Answering it means facing a harder question the pipeline has so far avoided. Every step has sorted documents into good and bad without ever saying what *good* means: Part 1 caught malformed text and removed duplicates, Part 3 scored surface form, yet none of them defined the quality they were chasing. This part confronts that gap first, then shows the practical answer a real pipeline uses.

By the end of Part 4, you will be able to:

- Explain why every **quality signal** is a cheap proxy for a judgment too expensive to run directly.
- Define what makes text **bad** from **good** (open, and settled only after training).

Bad is easy to spot, good is not

Defining bad text is straightforward. Bad text has signs you can count: it is too short, it is mostly symbols, it repeats one line, it is a duplicate of something you already kept. Part 1 caught all of these by counting characters, because a malformed document reveals itself in its shape.

Good text has no such signs. There is no measurement you can run on a single document that returns "this is worth learning from", because worth is *fitness for use* (Part 0): it depends on the goal you are training for and on what the model still needs to learn, neither of which is written on the page. So nobody filtering data ever truly writes down what good is. They pick a *proxy* for it: a property they can measure on each document, chosen because it tends to track quality even though it is not quality itself. The proxy that FineWeb introduced is **educational value**: how much genuine, learnable information a document carries, scored on a small 0-to-5 scale by a classifier trained on LLM-written ratings [4]. It was chosen for three reasons, in rising order of importance: it is intuitive to people, it is cheap enough to score across the whole web, and, above all, models trained on the data it selects do better on the benchmarks. Educational value is a deliberate choice, open to debate, which is why even trained human annotators disagree on the borderline cases [9].

This difference shapes the rest of the work. You can define bad up front and act on it cheaply. For good, you choose a proxy, apply it, and only find out whether the choice was sound much later, by training a model on the result and measuring how it does.

Every quality signal is a cheap proxy

Seen this way, the tools in Parts 1 to 3 share one idea. A **surface heuristic** is a cheap proxy for "is this well-formed?". An **LLM judge** is an expensive proxy for "would a careful reader call this good?". Each replaces a judgment we cannot make directly with a measurement we can.

Key takeaway

Every quality signal is a cheap proxy for an expensive judgment. The recipe is always the same: take the most trustworthy judgment you can afford, then distill it into a cheaper one, and repeat, each step trading a little accuracy for a large cut in cost, until the signal is cheap enough to run at the scale you need.

That recipe stacks the judges in tiers, three of them. It helps to start at the most trustworthy, most expensive tier and work down, since each tier exists only to make the one below it affordable.

The top tier is human judgment. The closest thing to a true measure of whether a document is worth learning from is a person reading it, the way an editor decides what belongs in a book. It is also the costliest: a qualified annotator spends minutes per document and is paid by the hour, so scoring runs into euros per document[11], against fractions of a cent for a model. Reading the roughly five billion surviving documents by hand is impossible, so people never judge the whole corpus. They score a small, curated **seed** instead, the same kind of high-trust curated data from Part 1.

[11] Illustrative: a qualified annotator costs on the order of tens of euros an hour and scores only a handful of documents in that time, so human scoring runs to euros per document, orders of magnitude above an LLM judge at about 0,0003 € per document. The machine prices behind that comparison, about 0,045 € per CPU-hour and about 3 € per GPU-hour with their sources, are in the cost footnotes of Part 3.

The middle tier is a large language model, calibrated against that seed. A capable model can read a document and rate it much as a person would, and the seed is how you check that it does: you keep the model whose scores track the human ones. But a model big enough to judge well is still far too expensive to run on the whole web, around **1,5 million €** for a single pass (Part 3). So it never sees every document either. It judges a larger *sample*, big enough to teach the next tier but far short of the full corpus.

The bottom tier is a small model that runs everywhere. The sample the large model judged becomes training data for a small, cheap model that reproduces its verdicts at a fraction of the cost, light enough to score every surviving document. Each tier, in short, trains or calibrates the one beneath it: people vouch for the large model, and the large model labels the data that teaches the small one.

This three-tier chain is what JQL (Judging Quality across Languages, pronounced *Jackal*) [9] builds; it came after OpenGPT-X and sharpens the same idea, and the details belong to the paper. The shape is always the same: pay one large cost once, on a sample, so a cheap step can run at scale.

Plot the tiers by cost and coverage and they fall on a single downward line: the cheaper a judgment is per document, the more documents it can afford to run on. The costliest tier, human review, reaches only a curated seed; the cheapest, surface filters, runs across the whole web:

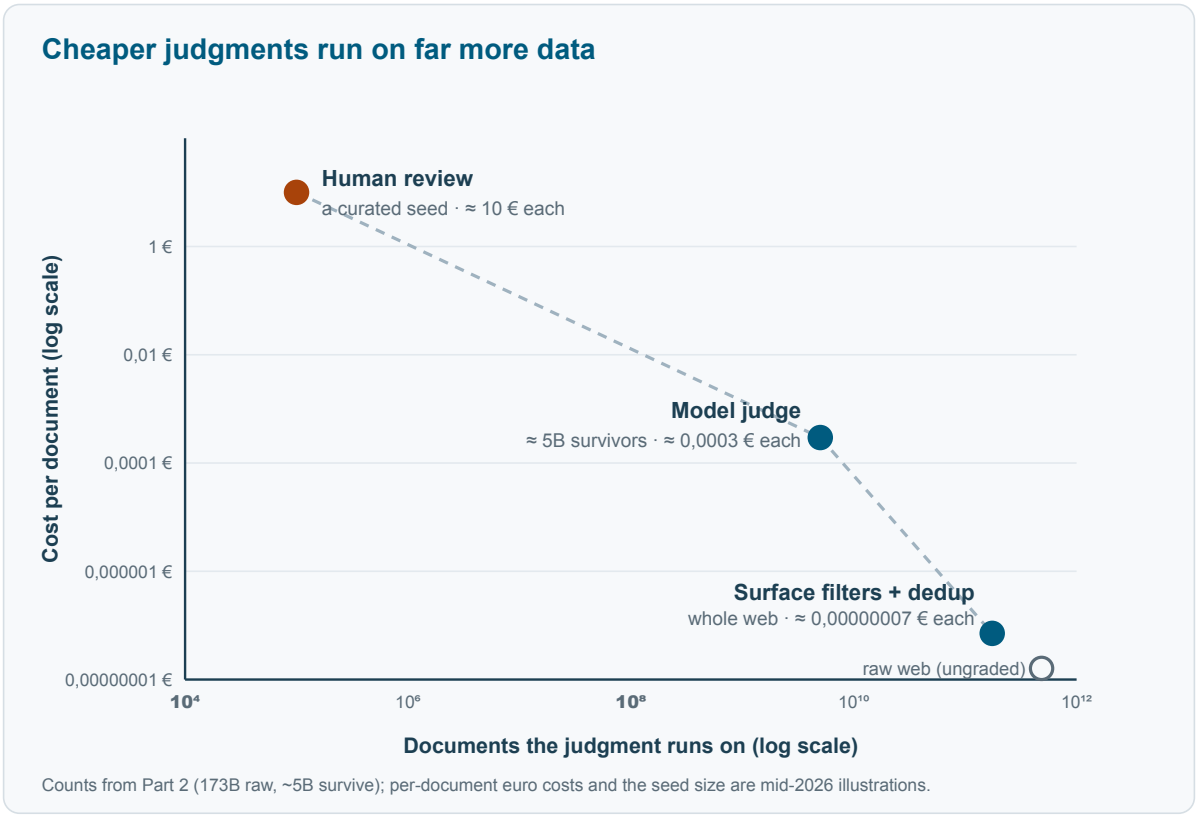


Figure 10: The judgment tiers on a cost-versus-coverage plot, both axes logarithmic. They fall on a downward line: human review costs about 10 € per document but reaches only a curated seed; the LLM judge costs about 0,0003 € and reaches the ~5 billion survivors; surface filters and deduplication cost about 0,00000007 € and run across the whole web. The raw web of 173 billion documents sits ungraded on the baseline.

Each tier down is cheaper than the one above, which invites one last question: could you take one more step and drop the model too? If some surface heuristic tracked the model's verdicts closely enough, you could filter with plain character counts and skip the reading entirely. Nobody has found one that does, and Part 3 is the reason to doubt one exists: surface form does not determine value, so a rule that only counts characters has nothing to read meaning with. Whether a rich enough set of surface features could ever line up perfectly with content is genuinely open; what we know is that none does

today. So in practice the chain bottoms out at a small model that reads what the words mean, and for now there is no cheaper tier beneath it.

The distilled model now puts a cheap value score on every surviving document, which turns the next step into a question: given a score for each document, where do you set the cutoff?

Filtering on rank rather than score

A proxy that runs on every language raises a practical problem the moment you try to filter with it. A model returns a quality *score*, and you keep documents above some cutoff. But the score's scale is arbitrary: one judge may rate most documents around 1, another around 4, while both rank documents in nearly the same order. An absolute cutoff like "keep everything scoring 3 or higher" then means something different for each judge and each language. The fix is to threshold on **rank**: keep the top share of documents, whatever their raw scores happen to be.

```
# Two quality judges score the same 10 documents from 0 to 5. They agree on the
# ranking but sit on different scales: "strict" clusters low, "generous" high.

strict   = [0.5, 0.8, 1.0, 1.2, 1.5, 1.8, 2.1, 2.4, 2.8, 3.2]
generous = [2.1, 2.6, 3.0, 3.3, 3.6, 3.9, 4.1, 4.3, 4.6, 4.9]

def keep_absolute(scores, cutoff=3.0):
    return sum(s >= cutoff for s in scores)

def keep_top_fraction(scores, fraction=0.4):
    # keep the best-scoring `fraction` of documents, whatever the scale
    ranked = sorted(scores)
    cut = ranked[int(len(scores) * (1 - fraction))]
    return sum(s >= cut for s in scores)

for name, scores in [("strict", strict), ("generous", generous)]:
    print(f"{name:8}  absolute >=3.0 keeps {keep_absolute(scores)}/10"
          f"   top 40% keeps {keep_top_fraction(scores)}/10")
```

```
strict    absolute >=3.0 keeps 1/10   top 40% keeps 4/10
generous  absolute >=3.0 keeps 8/10   top 40% keeps 4/10
```

The absolute cutoff keeps almost nothing from the strict judge and almost everything from the generous one, even though both ranked the documents the same way. The rank-based cutoff keeps the same share from each, so the filter means the same thing no matter which judge or which language produced the scores. It also gives you one clear control: the fraction you keep sets the trade-off between corpus size and average quality directly.

> ! Your turn
>
> Tightening the rank cutoff is only one way to raise the value of what you keep, and it always costs you corpus size. Name two *other* levers that improve the average quality of the surviving documents without simply keeping a smaller top share. Then consider the case OpenGPT-X actually used: a document is kept only if *three* independent judges each rank it in their own top share. Why would requiring agreement be safer than trusting one judge's ranking?

> **i** One way to reason about it
>
> Tightening the cutoff just moves along the quality-versus-quantity trade-off: a smaller corpus, higher average quality, nothing new gained. The other levers change the trade-off itself. **Source mixing**, upsampling high-value sources and downsampling low-value ones, lifts the average without judging any single document harder. And **combining signals of different kinds**, an LLM judge that reads meaning together with deterministic surface metrics, catches what either alone would miss: the judge scores value, the counts catch the malformed cases it overlooks. Requiring three judges to agree is a variation on that same idea: it guards against any one of them being wrong, since a single proxy has blind spots and noise, and a document that three differently-trained judges all rank highly is a safer choice than one that only a single model rated highly.

A wider family of proxies

Every proxy so far has done the same thing: score a whole document from its text, by its shape with a surface heuristic or by its meaning with an LLM judge, then keep it or drop it. Those are not the only instruments. Two more are worth knowing, because they show how broad the idea of a quality signal really is.

The first reads something other than the text at all. A document carries *implicit* signals from how the web treated it: how many pages link to it, where a search engine ranks it, how often people click or stay. These judge a page by its reception rather than its words, so they need neither a surface heuristic nor an LLM judge. OpenGPT-X did not use them, but they are a standing option.

The second does read the text, but asks a narrower question: how surprising is the text? To measure it, you can run a small language model over a document and measure its *perplexity*, how surprised the model is by the words: text that resembles the clean reference the model was trained on scores low, while gibberish, boilerplate, and spam score high.

Taken together, these signals are a reminder that quality can be guessed from almost anywhere, not only a literal reading of the words on the page.

What "good" still leaves open

JQL gives a strong, cheap proxy for good text and runs it across every language. It still leaves the definition of good itself open. Each step in this material has approximated good with something it could measure: Part 1 with well-formedness, and with whether a document added anything new or merely repeated what was already kept, Part 3 with those same surface heuristics, which it showed cannot reach value, and Part 4 with a model's value score. Underneath all of them, the final test of whether any of it worked is how the trained model does on a benchmark, which is itself a proxy for the quality we actually care about. The approximations stack, and at the bottom there is no measurement that reads "this text is good".

This is the honest shape of the problem. Defining good text well enough to measure it, and across every language at once, is open research, and the choice of proxy decides what kind of model the data will make.

Recap

Part 4 reframed the whole pipeline as a sequence of proxies. Bad text is definable and can be filtered cheaply; good text cannot be defined outright, so every quality signal is a cheap proxy for it, something measurable, and the only test of the choice comes downstream. Because a proxy good enough to read value is expensive to run, the move is to pay for it once on a sample and distill it into a cheap model that runs on the billions. Filtering on rank rather than raw score makes that proxy behave consistently across judges and languages. What remains open is the thing underneath: what makes text good at all.

Closing

This material set out to answer one question: where does the enormous amount of suitable text a language model needs come from, and what does *suitable* even mean. Four parts on, both halves have an answer. Where the text comes from is now concrete: curated and web sources, drawn along the two paths of Part 1 and reduced, cheaply and in the right order, to the clean fraction worth keeping. What *suitable* means turned out to be the harder finding: bad text can be defined up front by the tells you can count, but good text has none, so it can only be approximated by a proxy and confirmed after the fact.

Read back through the parts, here is what each was meant to leave you with:

Part	What you learned	How to apply it
1) The pipeline	Turn curated and web text into one clean, shared record, and keep that record *traceable*.	Save the metadata and intermediate outputs as you go.
2 Working at scale	At web scale every step is also a compute decision; wall-clock time, tied to the project's compute window, is the budget that binds.	Run the cheap, decisive cuts first and the expensive comparisons last.
3) The quality question	Cheap surface heuristics judge a document's *form* but stay blind to its *value*; reading value means paying for a model.	Never ask a surface filter whether text is *worth* learning from; it knows only the shape.
4) Cheap proxies for good text	Good text cannot be defined outright, so every quality signal is a cheap proxy you can confirm only after training.	Treat a quality score as a hypothesis to test downstream, not a ground truth to trust up front.

A language model is a product of its training data, so every decision taken before training, what to keep, what to call good, what to discard, is a decision about what the model can become. None of it shows in the finished weights, yet all of it is already there.

> **i** What we did not cover
>
> This material stops at preparing crawled and curated text. The biggest neighbor it leaves out is **synthetic data**, text generated by a model rather than collected, which sidesteps the crawl entirely but brings its own quality and diversity questions (see this survey of best practices). Tokenization and the *mixing weights* that set how much of each source a model sees are likewise their own topics.

A closing word on where to take this. The field moves quickly, and the specifics will date faster than the principles: the libraries turn over and the judges keep improving. What carries forward is the way of thinking, keep the data traceable, match each step to its budget, and treat every quality signal as a proxy you still have to check. Hold on to those, and the next tool will be easy to learn.

Bibliography

[1] International Organization for Standardization, "ISO/IEC 25012:2008 Software engineering — Software product Quality Requirements and Evaluation (SQuaRE) — Data quality model," Geneva, CH, Standard, 2008. [Online]. Available: https://www.iso.org/standard/35736.html

[2] R. Y. Wang and D. M. Strong, "Beyond Accuracy: What Data Quality Means to Data Consumers," *Journal of Management Information Systems*, vol. 12, no. 4, pp. 5–33, 1996, doi: 10.1080/07421222.1996.11518099.

[3] L. Gao *et al.*, "The Pile: An 800GB Dataset of Diverse Text for Language Modeling," *arXiv preprint arXiv:2101.00027*, 2020.

[4] G. Penedo *et al.*, "The FineWeb Datasets: Decanting the Web for the Finest Text Data at Scale," in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2024, pp. 30811–30849. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2024/hash/370df50ccfdf8bde18f8f9c2d9151bda-Abstract-Datasets_and_Benchmarks_Track.html

[5] N. Brandizzi *et al.*, "Data Processing for the OpenGPT-X Model Family," *arXiv preprint arXiv:2410.08800*, 2024.

[6] T. Jansen, Y. Tong, V. Zevallos, and P. O. Suarez, "Perplexed by quality: A perplexity-based method for adult and harmful content detection in multilingual heterogeneous web data," *arXiv preprint arXiv:2212.10440*, 2022.

[7] K. Lee *et al.*, "Deduplicating Training Data Makes Language Models Better," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Association for Computational Linguistics, 2022, pp. 8424–8445. doi: 10.18653/v1/2022.acl-long.577.

[8] J. Leveling *et al.*, "Evaluation of Document Deduplication Algorithms for Large Text Corpora," in *International Conference on Machine Learning, Optimization, and Data Science*, 2024, pp. 390–404.

[9] M. Ali *et al.*, "Judging Quality Across Languages: A Multilingual Approach to Pretraining Data Filtering with Language Models," in *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, 2025, pp. 8870–8909.