Chapter 10: Tokenization and Data at Scale

Learning Objectives

After reading this chapter, the reader should be able to:

1. Implement the Byte-Pair Encoding (BPE) algorithm from scratch, trace its merge operations on a sample corpus, and explain why subword tokenization solves the open-vocabulary problem that word-level models cannot handle.
2. Compare BPE, WordPiece, Unigram, and SentencePiece tokenization methods in terms of their algorithmic approach, vocabulary construction, and impact on downstream model performance.
3. Analyze how tokenization choices affect model behavior across languages, domains, and efficiency metrics, including fertility (tokens per word), compression ratio, and coverage of rare and morphologically rich words.
4. Describe the key stages of a modern data curation pipeline – web crawling, deduplication, quality filtering, data mixing – and explain how each stage affects pre-training quality.

In Chapter 9, we examined three pre-training paradigms – BERT's masked language modeling, GPT's causal language modeling, and T5's span corruption – that train Transformers on massive text corpora to produce models with transferable linguistic knowledge. We treated "tokens" as given and focused on what the model does with them. But we left a question dangling in plain sight: what *is* a token? When BERT masks 15% of tokens, what exactly are those tokens? When GPT predicts the next token, what is the unit of prediction – a word, a character, a syllable, something else entirely? The answer to this question is not a minor preprocessing detail. It is a design decision that cascades through every aspect of model behavior: the size of the vocabulary, the length of the input sequence, the computational cost of self-attention (which, as we established in Section 8.2, scales quadratically with sequence length), and even which languages the model serves well and which it quietly penalizes. A language model can only predict what its tokenizer can represent. If the tokenizer has no entry for "Kuenstliche" but knows "the" as a single unit, the model is already biased before a single gradient is computed.

This chapter opens the black box. We begin with the fundamental tension between word-level and character-level representations and explain why subword tokenization emerged as the principled resolution (Section 10.1). We then present Byte-Pair Encoding in full algorithmic detail, including the byte-level variant that powers GPT-2 through GPT-4 (Section 10.2). We introduce SentencePiece and the Unigram language model as alternatives that bring language independence and probabilistic flexibility (Section 10.3). We analyze how tokenization choices ripple through model performance, multilingual fairness, and the emerging push toward tokenizer-free architectures (Section 10.4). Finally, we turn to the other hidden engine of language model quality – the data curation pipeline that transforms petabytes of raw web crawls into the carefully balanced corpora that these models actually learn from (Section 10.5). The unifying thread, established in Chapter 1, remains prediction: tokenization defines *what* the model predicts, and data curation determines *what patterns* are available to learn.

10.1 From Words to Subwords

How should a model handle a word it has never seen before?

This question, which seems almost trivially simple, turns out to be one of the most consequential design choices in modern natural language processing. The answer that the field settled on – subword tokenization – took decades to emerge and draws on ideas from data compression, computational morphology, and information theory. We begin by understanding why the two extremes (whole words and individual characters) both fail, and why the middle ground succeeds.

10.1.1 The Open-Vocabulary Problem

Every language model must maintain a vocabulary V: a finite set of tokens that the model can read, embed, and predict. In word-level tokenization, each distinct word type in the training corpus receives its own entry in V. The model learns an embedding vector for "cat," another for "running," another for "photosynthesis," and so on. This approach is clean, intuitive, and was standard practice through the early neural NLP era. It also harbors a fatal flaw that becomes visible only at scale. Natural language has an effectively unbounded vocabulary. New words enter the language continuously – neologisms ("doomscroll," "hallucinate" in the LLM sense), technical terms ("CRISPR-Cas9"), proper nouns ("ChatGPT"), morphological variants ("ungooglable"), and compound words in languages like German ("Donaudampfschifffahrtsgesellschaft"). The frequency distribution of words follows Zipf's law: a small number of types account for a large fraction of tokens, while the remaining types form a long tail of rare words that collectively matter enormously. The most frequent 10,000 English word types cover approximately 90% of typical text, but the remaining 10% would require hundreds of thousands of additional entries to capture. For a vocabulary of $|V| = 50{,}000$ words, approximately 5% of tokens in web text will be out-of-vocabulary (OOV), and for specialized domains such as biomedical or legal text, OOV rates can exceed 15%. Every OOV token is mapped to a single `[UNK]` symbol, destroying all distinction between unknown words – the model cannot tell whether `[UNK]` was a person's name, a chemical compound, or a typo.

A natural instinct is to solve this problem by making the vocabulary larger. Doubling from 50K to 100K entries captures a few additional percent of coverage, but at a steep cost: the embedding matrix has $|V| \times d$ parameters, where d is the embedding dimension (typically 768 to 4096), meaning that a move from 50K to 100K vocabulary adds tens of millions of parameters to the model while providing only marginal coverage improvement. Zipf's law makes the diminishing returns severe: each additional thousand entries yields less coverage than the previous thousand. Worse, the rare words admitted into a large vocabulary receive few training examples, so their embeddings remain poorly estimated. The open-vocabulary problem is not solvable by brute force. A fundamentally different representational strategy is needed.

10.1.2 Word-Level vs. Character-Level Trade-offs

If word-level tokenization fails because vocabularies cannot be large enough, the opposite extreme – character-level tokenization – offers an appealingly clean solution. The English alphabet has 26 lowercase letters, 26 uppercase letters, 10 digits, and a handful of punctuation marks. A character-level vocabulary of roughly 100–300 symbols covers any English text with zero OOV tokens. Any word, no matter how rare or novel, can be represented as a sequence of characters. The problem is what this representation costs. The sentence "The cat sat on the mat" requires 6 tokens at the word level but 22 characters (including spaces). For the Transformer architecture introduced in Chapter 8, this difference is not merely an inconvenience: self-attention computes a $T \times T$ attention

matrix, making its cost quadratic in sequence length T. A factor of roughly 4–5 in sequence length translates to a factor of 16–25 in attention computation. A 512-word document that fits comfortably in a word-level model's context window would require approximately 2,500 character tokens, pushing against or exceeding the context limits of many architectures and dramatically increasing training time.

Beyond computational cost, character-level models face a representational challenge. A word-level model starts with the assumption that "cat" is a meaningful unit and learns to embed it. A character-level model starts with "c," "a," "t" and must learn from scratch that this particular sequence forms a meaningful unit, that it refers to a small feline, and that it behaves syntactically as a noun. The model must reconstruct the entire morphological and semantic hierarchy of language from raw characters, which requires more data, more parameters, and more training time. Empirical results bear this out: character-level Transformers can achieve competitive performance on some tasks, but they require significantly more computational resources and are generally inferior to subword models on standard benchmarks when matched for compute budget. The reading-speed analogy is apt here. Fluent readers do not process text letter by letter; they recognize common words as whole units and slow down only for unfamiliar terms, effectively performing a biological version of subword tokenization. A purely character-level model forces the machine equivalent of letter-by-letter reading on every word, including the ones it has seen millions of times.

10.1.3 The Subword Hypothesis

The solution that the field converged on – and that now powers every major language model – is subword tokenization: represent common words as single tokens and decompose rare words into sequences of smaller, reusable units. The word "unbelievable" might be represented as `["un", "believ", "able"]`, "transformers" as `["transform", "ers"]`, and "cat" simply as `["cat"]`. This achieves three properties simultaneously. First, it provides *open vocabulary*: any word, no matter how novel, can be composed from subword pieces drawn from a finite vocabulary, eliminating the OOV problem entirely. Second, it produces *compact sequences*: common words remain single tokens, keeping sequences short and attention computation manageable. Third, it enables *meaningful decomposition*: when the algorithm works well, subword boundaries often align with morphological boundaries, so the model can share knowledge between "run," "running," "runner," and "runs" through their common prefix – a property we first glimpsed in Section 4.4.2, where FastText's character n-grams allowed word embeddings to generalize across morphological variants. Think of it as LEGO: a modest collection of reusable blocks can build any structure, from simple walls to elaborate castles. Common structures come as pre-assembled pieces for efficiency, while novel creations are assembled from basic blocks on demand.

Input sentence: “The unbelievable transformer model”

Word-level	The · unbelievable · transformer · model	*4 tokens*
BPE (32K)	The · un · believ · able · transform · er · model	*7 tokens*
WordPiece	The · un · ##believe · ##able · transform · ##er · model	*7 tokens*
Unigram	The · un · believe · able · trans · former · model	*7 tokens*

Subword methods (BPE, WordPiece, Unigram) decompose rare words into reusable pieces while keeping common words intact.

– 4

tokens; (2) BPE (32K vocab): [“The”, “un”, “believ”, “able”, “transform”, “er”, “model”] – 7 tokens; (3) Unigram: [“The”, “un”, “believe”, “able”, “trans”, “former”, “model”] – 7 tokens with slightly different boundaries; (4) character-level: [“T”, “h”, “e”, ” “,”u”, “n”, “b”, ...] – 34 tokens. Each row is color-coded, with subword boundaries marked by vertical bars. Dimensions: 16cm x 8cm. Tool: TikZ.]

The practical question that determines a subword tokenizer’s behavior is the vocabulary size $|V|$. At one extreme, $|V| = 256$ (raw bytes) produces near-character-level sequences. At the other extreme, $|V| = 200{,}000$ approaches word-level with its OOV vulnerabilities. The sweet spot for modern large language models falls in the range of 32,000 to 128,000 tokens: GPT-2 uses 50,257, LLaMA uses 32,000, and GPT-4 uses approximately 100,277. This range balances sequence compactness against vocabulary coverage and embedding parameter cost, though the optimal value depends on model size, training data, and the target language distribution. Whether the sweet spot is 32K or 64K or somewhere else remains, frankly, an empirical question with no clean theoretical answer – practitioners tune it alongside other hyperparameters and accept the result.

The subword approach originated not in NLP but in data compression. The algorithm we present next – Byte-Pair Encoding – was invented by Philip Gage in 1994 as a method for compressing text files [Gage, 1994]. Two decades later, Sennrich, Haddow, and Birch [Sennrich et al., 2016] recognized that the same greedy merging procedure that compresses text also produces an excellent vocabulary for neural machine translation. That insight – that good compression implies good tokenization – turned out to be one of the most impactful ideas in modern NLP infrastructure.

10.2 Byte-Pair Encoding

If we let the data decide which character sequences matter most, what vocabulary does it build?

Byte-Pair Encoding (BPE) is the dominant tokenization algorithm in modern language modeling. GPT-2, GPT-3, GPT-4, LLaMA, and most other major language models use some variant of BPE to construct their vocabularies. The algorithm is elegant in its simplicity: begin with characters, count which pairs of adjacent symbols appear most often, merge the most frequent pair into a

new symbol, and repeat. The result is a vocabulary that adapts to the statistical structure of the training corpus, capturing frequent character sequences as single tokens and leaving rare sequences decomposed into smaller pieces.

10.2.1 The BPE Algorithm Step by Step

The BPE algorithm proceeds as follows. We start with a corpus of words, each represented as a sequence of characters plus a special end-of-word marker `</w>`. The initial vocabulary V_0 contains all unique characters that appear in the corpus. At each iteration, we count the frequency of every adjacent pair of symbols across the entire corpus and merge the most frequent pair into a new single symbol. Formally, the merge operation selects the pair that maximizes the co-occurrence count:

$$\text{pair}^* = \arg\max_{(a,b)\in\text{pairs}(V)} \text{count}(a, b) \tag{10.1}$$

After merging, all instances of the adjacent pair (a, b) in the corpus are replaced by the new symbol ab, and ab is added to the vocabulary. The vocabulary grows by exactly one token per merge. After K merges, the vocabulary contains $|V_0| + K$ tokens. The algorithm terminates when the desired vocabulary size is reached.

We walk through this example concretely. Consider a small corpus with five word types and their frequencies: "l o w " appears 5 times, "l o w e r " appears 2 times, "n e w " appears 6 times, "n e w e r " appears 3 times, and "w i d e r " appears 2 times. The initial vocabulary is {d, e, i, l, n, o, r, s, w, }. In the first iteration, we count all adjacent pairs: (l, o) has frequency 7 (from "low" and "lower"), (o, w) has frequency 7, (e, r) has frequency 7 (from "lower," "newer," and "wider"), (r,) has frequency 7, (n, e) has frequency 9, (e, w) has frequency 9, and so on. The pair (n, e) and (e, w) both have frequency 9, so we select one – say (e, r) appears 7 times. Actually, let us recount more carefully: (n, e) appears in "new" (6 times) and "newer" (3 times), giving 9. We select (e, w) with frequency 9 as the highest pair (it appears in "new" 6 times and "newer" 3 times). The pair is merged into "ew," the corpus is updated, and "ew" is added to the vocabulary. In the second iteration, pair frequencies are recomputed on the updated corpus, and the next most frequent pair is merged. Each merge captures a frequently recurring character combination, and over many iterations, the vocabulary grows from characters to character bigrams to common subwords to frequent whole words.

The following implementation demonstrates BPE from scratch in Python, using only the `collections.Counter` class and a regular expression for merge application:

```python
import re
from collections import Counter

def get_pairs(vocab):
    """Count adjacent symbol pairs across all words."""
    pairs = Counter()
    for word, freq in vocab.items():
        symbols = word.split()
        for i in range(len(symbols) - 1):
            pairs[(symbols[i], symbols[i + 1])] += freq
    return pairs

# Training corpus: each word as space-separated characters + end marker
```

BPE Merge Steps

Step	Merge Pair (freq)	Corpus after merge
Init	Vocab: {d, e, i, l, n, o, r, w, </w>} Corpus: "l o w </w>":5 "l o w e r </w>":2 "n e w </w>":6 "n e w e r </w>":3 "w i d e r </w>":2	
1	**(e, w) → ew** freq = 9	"l o w </w>":5 "l o w e r </w>":2 "n ew </w>":6 "n ew e r </w>":3 "w i d e r </w>":2
2	**(e, r) → er** freq = 7	"l o w </w>":5 "l o w er </w>":2 "n ew </w>":6 "n ew er </w>":3 "w i d er </w>":2
3	**(er, </w>) → er</w>** freq = 7	"l o w </w>":5 "l o w er</w>":2 "n ew </w>":6 "n ew er</w>":3 "w i d er</w>":2
4	**(n, ew) → new** freq = 9	"l o w </w>":5 "l o w er</w>":2 "new </w>":6 "new er</w>":3 "w i d er</w>":2
5	**(l, o) → lo** freq = 7	"lo w </w>":5 "lo w er</w>":2 "new </w>":6 "new er</w>":3 "w i d er</w>":2
6	**(lo, w) → low** freq = 7	"low </w>":5 "low er</w>":2 "new </w>":6 "new er</w>":3 "w i d er</w>":2

Final vocab (+6): {d, e, i, l, n, o, r, w, </w>, ew, er, er</w>, new, lo, low}

Figure 1: Figure 10.2 – BPE Step-by-Step Merge Example

```python
corpus = {"l o w </w>": 5, "l o w e r </w>": 2, "n e w </w>": 6,
          "n e w e r </w>": 3, "w i d e r </w>": 2}

print("Initial vocabulary:", set(c for w in corpus for c in w.split()))
for step in range(10):  # 10 BPE merges
    pairs = get_pairs(corpus)
    if not pairs:
        break
    best = max(pairs, key=pairs.get)
    print(f"Merge {step+1}: {best[0]}+{best[1]} -> {''.join(best)}"
          f" (freq={pairs[best]})")
    # Apply merge to entire corpus
    merged = re.escape(' '.join(best))
    replacement = ''.join(best)
    corpus = {re.sub(merged, replacement, w): f
              for w, f in corpus.items()}
# Result: vocabulary grows from characters to meaningful subwords
```

One subtlety that students sometimes miss: BPE only considers *adjacent* pairs. The characters "l" and "w" in "low" are merged not because both are individually frequent but because they appear next to each other frequently. This adjacency constraint guarantees that every merged symbol corresponds to a contiguous substring of the original text – a property that would break if we considered arbitrary character co-occurrences.

10.2.2 Byte-Level BPE (GPT-2 Variant)

Standard BPE begins from Unicode characters, which creates an immediate complication: Unicode defines over 140,000 characters across the world's writing systems. A character-level base vocabulary for Chinese alone would contain thousands of entries before any merges occur, and handling mixed-script text requires language-specific pre-tokenization rules (splitting on whitespace for English, using segmenters for Chinese and Japanese, handling Arabic morphology). Radford et al. [2019] proposed an elegant solution for GPT-2: start BPE not from Unicode characters but from raw bytes. Every text file, regardless of language or encoding, is ultimately a sequence of bytes with values from 0 to 255. The base vocabulary is therefore exactly 256 tokens – universal, compact, and requiring no language-specific preprocessing whatsoever. BPE merges then build up from bytes to subwords to words in exactly the same manner as character-level BPE, but the starting point is language-agnostic.

The trade-off is sequence length. Under the UTF-8 encoding used by virtually all modern text, ASCII characters (which cover English letters, digits, and common punctuation) occupy one byte each, but characters outside the ASCII range require two to four bytes. A Chinese character typically requires three bytes, meaning that byte-level BPE starts with sequences roughly three times longer for Chinese text than for English text, even before any merges are applied. With a sufficient number of merges, this gap narrows substantially – common Chinese characters are quickly merged into single tokens – but it never fully closes. The universality of the approach was deemed worth the cost: GPT-2 (50,257 tokens), GPT-3, GPT-4 (100,277 tokens), and LLaMA all adopted byte-level BPE. The practical implication is that these models can handle any text – English, Chinese, Arabic, emoji, code, mathematical notation – without ever encountering an unknown byte.

GPT-2's implementation also introduced regex-based pre-tokenization to prevent merges from crossing word boundaries. Before BPE merges are counted, the text is split into chunks using a regular expression pattern that separates words, numbers, and punctuation. This prevents the space at the end of "the" from merging with the "c" at the start of "cat" – a merge that would produce a useless token spanning a word boundary. This pre-tokenization step is a practical optimization, not a fundamental change to the algorithm, but it significantly improves the quality of the resulting vocabulary by ensuring that most tokens correspond to complete words or meaningful subword units rather than accidental cross-word fragments.

10.2.3 Implementation and Practical Considerations

Once a BPE model has been trained – that is, once the ordered list of merge rules has been determined from the training corpus – encoding new text at inference time requires applying those merge rules in the same priority order. The procedure starts by decomposing the input into its base units (bytes or characters), then iteratively applying the highest-priority applicable merge until no more merges can be applied. The result is a deterministic tokenization: the same input text always produces exactly the same sequence of tokens. This determinism is a distinctive property of BPE that we will contrast with the Unigram model in Section 10.3, which can produce multiple valid tokenizations of the same input.

The merge table – the ordered list of approximately K merge rules, where K is the vocabulary size minus the base vocabulary size – must be stored alongside the vocabulary and distributed with the model. For GPT-2 with 50,257 tokens and a base vocabulary of 256 bytes, the merge table contains roughly 50,000 entries, each specifying a pair of symbols and their merged result. This is a modest storage requirement (a few megabytes) but a critical artifact: without the merge table, one cannot reproduce the tokenization.

Speed was historically a concern for BPE encoding, since the naive algorithm requires iterating through the merge table for each input token. Modern implementations have eliminated this bottleneck entirely. OpenAI's *tiktoken* library, written in Rust with Python bindings, uses compiled regular expressions and optimized data structures to achieve encoding speeds of millions of tokens per second. The Hugging Face *tokenizers* library similarly uses a Rust backend and can tokenize an entire book in under a second. The tokenizer is never the bottleneck in modern language model training or inference – the forward pass through the Transformer is orders of magnitude more expensive. The practical advice for practitioners is straightforward: use an established tokenizer library (tiktoken for OpenAI models, Hugging Face tokenizers for open-source models, SentencePiece for models that use it) rather than implementing BPE from scratch. The from-scratch implementation above is pedagogically valuable for understanding the algorithm, but production use demands the optimized implementations.

Sidebar: WordPiece – BPE's Close Cousin

WordPiece, developed at Google and used in BERT and its derivatives [Schuster and Nakajima, 2012], follows the same iterative merging logic as BPE but differs in the criterion for selecting which pair to merge. Where BPE selects the pair with the highest raw frequency – $\text{pair}^* = \arg\max \text{count}(a, b)$ – WordPiece selects the pair that maximizes the *likelihood* of the training corpus under a unigram language model. Concretely, WordPiece merges the pair (a, b) that maximizes $\frac{p(ab)}{p(a) \cdot p(b)}$, where $p(\cdot)$ denotes the relative frequency of a symbol. This ratio is equivalent to pointwise mutual information (PMI) and favors pairs whose co-occurrence is surprising relative to their individual frequencies,

rather than pairs that are simply common. In practice, the vocabularies produced by BPE and WordPiece are very similar for the same vocabulary size, and the performance difference on downstream tasks is generally negligible. The more consequential difference is architectural: WordPiece marks continuation tokens with a "##" prefix (e.g., "playing" becomes ["play", "##ing"]), while BPE typically marks word-initial tokens with a special prefix character. We do not cover WordPiece in further detail here, as BPE has become the dominant standard for generative language models, but the reader should recognize the "##" prefix when working with BERT-family models.

10.3 SentencePiece and Unigram

Can we build a tokenizer that handles any language without knowing anything about that language?

BPE, as described in Section 10.2, assumes that the input text has already been split into words – typically by whitespace. This assumption works tolerably well for English, German, and other space-delimited languages, but it fails entirely for Chinese, Japanese, and Thai, which do not use spaces between words. Even for space-delimited languages, the whitespace assumption introduces language-specific decisions: should contractions like "do not" (often written "dont" without apostrophe in raw text) be split? Should hyphenated compounds be treated as one word or two? SentencePiece and the Unigram language model tokenizer were developed to eliminate these assumptions, creating a tokenization framework that is genuinely language-independent.

10.3.1 SentencePiece: Language-Independent Preprocessing

SentencePiece, introduced by Kudo and Richardson [2018], resolves the language-dependence problem by treating the entire input as a raw stream of Unicode characters, with no pre-tokenization step whatsoever. Whitespace is not used as a delimiter; instead, it is treated as just another character – specifically, it is replaced by a special underline marker "_" (Unicode U+2581) so that tokenization is fully reversible. The input sentence "The cat sat" becomes "_The_cat_sat" before any segmentation algorithm is applied. Because the underline is an explicit character in the vocabulary, the original whitespace can be perfectly reconstructed from the tokenized output – a property called *lossless tokenization* that is essential for generative models that must produce properly formatted text.

The key insight behind SentencePiece is that it separates the *preprocessing framework* from the *segmentation algorithm*. SentencePiece itself is not a tokenization algorithm; it is a wrapper that handles language-independent preprocessing and then applies either BPE or Unigram as its internal segmentation method. When using BPE mode, SentencePiece applies the standard BPE merge algorithm described in Section 10.2 but operates on the whitespace-normalized character stream rather than on pre-split words. When using Unigram mode (described in Section 10.3.2), it applies the probabilistic Unigram segmentation. This separation is critical because it means the same preprocessing pipeline handles English, Chinese, Japanese, Arabic, Hindi, and code without any language-specific rules. The algorithm does not need to know whether the input language uses spaces, what its morphological structure is, or how punctuation works – it simply processes the raw character stream. T5, LLaMA, XLNet, and ALBERT all adopted SentencePiece, and its language-independence has made it the preferred framework for multilingual models. The following code demonstrates training both BPE and Unigram models through the SentencePiece library:

```python
import sentencepiece as spm
import tempfile, os

# Write a small training corpus
corpus_text = ("The cat sat on the mat. "
               "Language models predict the next word. "
               "Tokenization splits text into subwords. " * 100)
corpus_file = os.path.join(tempfile.gettempdir(), "corpus.txt")
with open(corpus_file, "w") as f:
    f.write(corpus_text)

# Train BPE and Unigram tokenizers via SentencePiece
for model_type in ["bpe", "unigram"]:
    prefix = os.path.join(tempfile.gettempdir(), f"sp_{model_type}")
    spm.SentencePieceTrainer.train(
        input=corpus_file, model_prefix=prefix,
        vocab_size=100, model_type=model_type)
    sp = spm.SentencePieceProcessor(model_file=f"{prefix}.model")
    text = "Tokenization determines what the model predicts"
    pieces = sp.encode(text, out_type=str)
    print(f"{model_type:>8s}: {pieces}")
# BPE and Unigram produce different segmentations of the same text
```

An important practical note: SentencePiece is often confused with the Unigram algorithm itself. Students sometimes say "we used a SentencePiece tokenizer" when they mean "we used the Unigram algorithm within SentencePiece." The distinction matters because SentencePiece with BPE mode and SentencePiece with Unigram mode produce genuinely different vocabularies and segmentations. The framework is the same; the algorithm inside is not.

10.3.2 The Unigram Language Model Tokenizer

The Unigram language model tokenizer [Kudo, 2018] takes a fundamentally different approach from BPE. Where BPE builds a vocabulary *bottom-up* – starting with characters and iteratively merging pairs – the Unigram model works *top-down*: it begins with a large initial vocabulary containing all substrings up to a specified length and iteratively *removes* the tokens that contribute least to the overall likelihood of the corpus. The Unigram model is probabilistic: it assigns a probability $p(x_i)$ to each subword token x_i in the vocabulary, and the probability of a particular segmentation $\mathbf{x} = (x_1, x_2, \ldots, x_{|\mathbf{x}|})$ is the product of individual token probabilities under a unigram (independence) assumption:

$$P(\mathbf{x}) = \prod_{i=1}^{|\mathbf{x}|} p(x_i) \tag{10.2}$$

Given a word w, there are typically many valid segmentations – the set $S(w)$ of all ways to decompose w into sequences of subwords from the vocabulary. The optimal segmentation is the one that maximizes the log-likelihood:

$$\mathbf{x}^* = \arg\max_{\mathbf{x} \in S(w)} \sum_{i=1}^{|\mathbf{x}|} \log p(x_i) \tag{10.3}$$

This optimal segmentation is computed efficiently using the Viterbi algorithm over a lattice of possible subword decompositions, exactly analogous to the Viterbi decoding used in hidden Markov models (Section 3.3 in Chapter 3). The key computational insight is that the lattice structure allows dynamic programming: the best segmentation ending at each character position can be computed in linear time by considering all vocabulary tokens that end at that position.

Training the Unigram model proceeds through an expectation-maximization (EM) procedure. The algorithm initializes with a large vocabulary (for instance, all substrings of the corpus up to length 10 or 20, plus all individual characters), estimates subword probabilities via EM, and then prunes the vocabulary by removing the 10–20% of tokens whose removal causes the smallest increase in the overall corpus loss. This prune-and-re-estimate cycle repeats until the vocabulary reaches the desired size. The result is a vocabulary optimized for *likelihood* rather than *frequency*: the retained tokens are those that are collectively most useful for compactly representing the corpus, not merely those that appear most often as adjacent pairs.

The difference between BPE's frequency-based merging and Unigram's likelihood-based pruning leads to different vocabularies with subtly different properties. BPE tends to produce tokens that reflect the most common character sequences, which may or may not align with morphological boundaries. Unigram tends to retain tokens that have high information content – tokens that are genuinely useful for distinguishing between different words. An analogy that captures the difference: BPE builds a vocabulary the way a child learns to read, first recognizing frequent letter combinations and gradually building up to common words. Unigram builds a vocabulary the way an editor curates a dictionary, starting with an exhaustive list of candidate entries and removing those that add the least value.

10.3.3 Subword Regularization and Its Benefits

The probabilistic nature of the Unigram model enables a training technique that BPE's determinism cannot support: *subword regularization* [Kudo, 2018]. Because the Unigram model assigns probabilities to all valid segmentations of a word, not just the single best one, it can *sample* different segmentations during training rather than always using the optimal Viterbi segmentation. In one training epoch, the word "unbelievable" might be segmented as `["un", "believ", "able"]`; in the next epoch, the same word might appear as `["unbeliev", "able"]` or `["un", "believe", "able"]`. Each segmentation is drawn from the distribution defined by Equation 10.2, with more probable segmentations sampled more frequently.

This stochastic tokenization acts as a powerful form of data augmentation. By varying the segmentation boundaries, the model is forced to learn representations that are robust to the precise placement of subword breaks rather than memorizing specific tokenizations. The effect is analogous to image augmentation techniques (random cropping, rotation, color jitter) in computer vision, where showing the model different views of the same object during training improves generalization. Empirical results validate the approach: Kudo [2018] reported improvements of 0.5–1.5 BLEU points on machine translation benchmarks from subword regularization alone, with the gains coming entirely from training-time stochasticity. At inference time, the deterministic Viterbi segmentation is used to ensure reproducibility – the stochasticity is a training-only technique.

An interesting development that bridges the gap between BPE and Unigram is BPE-Dropout [Provilkov et al., 2020]. This method introduces subword regularization to BPE by randomly skipping merge operations during training: at each training step, each merge in the BPE merge table is applied with probability $1 - p$ and skipped with probability p (where p is typically set to 0.1). The

effect is that common merges are occasionally "undone," producing segmentations that are more fragmented than the standard BPE output but that expose the model to diverse subword boundaries. BPE-Dropout achieved improvements comparable to Unigram-based subword regularization on several machine translation benchmarks, suggesting that the benefit comes from the stochasticity itself rather than from the specific probabilistic framework. The practical recommendation is clear: if training with a BPE tokenizer, enable BPE-Dropout; if training with a Unigram tokenizer via SentencePiece, enable sampling-based regularization. Deterministic tokenization should be reserved for inference, where reproducibility is paramount.

10.4 The Impact of Tokenization

Does the tokenizer really matter, or is it just a preprocessing step?

We have spent considerable space describing *how* tokenizers work. We now turn to the question of *why it matters*: how does the choice of tokenizer – its algorithm, vocabulary size, and training data – affect the model that sits downstream of it? The short answer is that it matters enormously, and in ways that are not always obvious. Tokenization determines the model's vocabulary, the length of its input sequences, the computational cost of every forward pass, which languages it handles gracefully, and which it effectively discriminates against. A model is no more multilingual than its tokenizer.

10.4.1 Vocabulary Size vs. Sequence Length

Vocabulary size and average sequence length are locked in an inverse relationship that stems directly from the information-theoretic structure of language. A larger vocabulary means more words and subwords are represented as single tokens, producing shorter sequences for any given text. A smaller vocabulary means more words must be decomposed into multiple tokens, producing longer sequences. The computational implications are significant: for a Transformer with context length L, the self-attention mechanism requires $\mathcal{O}(L^2)$ time and memory, making sequence length a first-order concern for both training cost and inference latency (a constraint that becomes especially acute for the long-context models discussed in Chapter 14).

The quantitative trade-off follows a characteristic curve. For a BPE tokenizer trained on English text, going from a vocabulary of 1,000 to 10,000 tokens reduces the average number of tokens per word from approximately 2.8 to 1.4 – a dramatic improvement. Going from 10,000 to 32,000 reduces it further to approximately 1.15. Going from 32,000 to 128,000 yields only a marginal further reduction to approximately 1.05. This diminishing-returns curve is a direct consequence of Zipf's law: the first few thousand vocabulary entries capture the most frequent words, which account for the bulk of text, and each subsequent batch of entries captures progressively rarer words that appear infrequently. On the cost side, the embedding matrix has $|V| \times d$ parameters and the output softmax layer has $d \times |V|$ parameters, meaning that vocabulary size directly increases the model's parameter count and memory footprint. For an embedding dimension of $d = 4096$ (typical for models in the 7B–70B parameter range), going from 32K to 128K vocabulary adds approximately 400 million parameters to embeddings alone – a non-trivial fraction of a smaller model's total parameter budget. The numbers from practice reflect this balancing act: GPT-2 settled on 50,257 tokens, LLaMA chose 32,000, GPT-4 expanded to roughly 100,277, and the recent Llama 3 family uses 128,000. These are not arbitrary choices but reflect each team's judgment about the optimal trade-off between sequence compactness and parameter efficiency for their specific model size and training data.

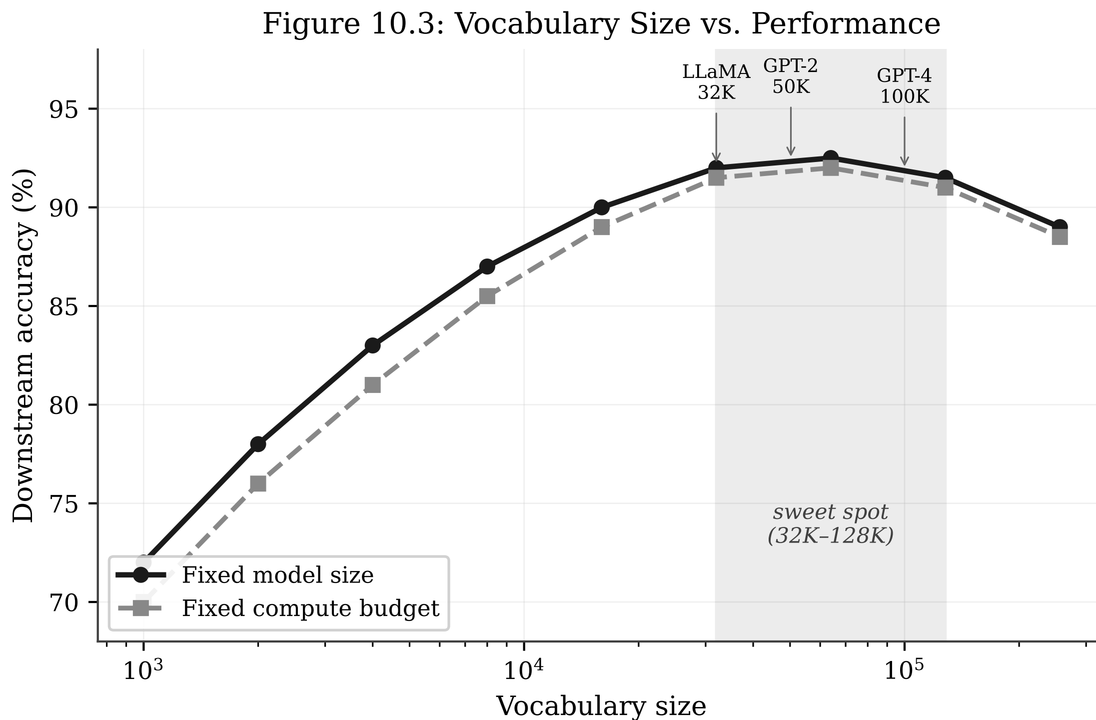

Figure 2: Figure 10.3 – Vocabulary Size vs

10.4.2 Multilingual Tokenization and the Fertility Problem

The consequences of tokenizer design become starkest when we look across languages. *Fertility* measures how many subword tokens a tokenizer produces per whitespace word (or per semantic unit for languages without whitespace boundaries):

$$\text{fertility}(w) = \frac{\text{number of subword tokens for } w}{\text{number of whitespace words in } w} \tag{10.4}$$

A fertility of 1.0 means each word is represented by a single token – the ideal case. A fertility of 3.0 means each word requires on average three tokens, tripling the effective sequence length and, consequently, the computational cost, the API charges (which are priced per token), and the amount of context window consumed per unit of meaning. For GPT-2's tokenizer, which was trained predominantly on English web text, English has a fertility of approximately 1.0–1.3. German, with its compound words and inflectional morphology, reaches 1.5–2.0. Chinese, where each character is typically encoded as multiple bytes under UTF-8 and the tokenizer has relatively few Chinese-specific merges, reaches 3.0–4.0 tokens per semantic unit. Arabic and Hindi fare similarly poorly, and Petrov et al. [2023] measured fertility ratios ranging from 1.0 (English) to an extraordinary 15.0 (Myanmar) across 17 languages.

The cause is straightforward: BPE merges are frequency-driven, and frequency in the training corpus is dominated by whatever language constitutes the majority of the data. For GPT-2, the training data (WebText) was predominantly English, so the most frequent byte pairs – and therefore the most aggressive merges – correspond to English character sequences. The merges that would compress Chinese, Arabic, or Hindi text efficiently simply never rose to the top of the frequency ranking. The result is a *tokenization tax*: non-English users pay more per semantic unit in computational cost,

API pricing, and consumed context window. A 4,096-token context window holds roughly 3,000 English words but only 800–1,000 Chinese semantic units, effectively giving Chinese users one-third the working memory. Petrov et al. [2023] documented this systematically and framed it as an equity issue – which it is.

This problem is not inherent to subword tokenization. It is a consequence of training data imbalance, not of the algorithm itself. The BLOOM project [BigScience, 2022] demonstrated this directly: by training its tokenizer on a deliberately balanced multilingual corpus covering 46 languages, BLOOM achieved substantially more equitable fertility across languages. The lesson is that tokenizer training data composition matters as much as model training data composition. A model that aspires to multilingual capability needs a tokenizer trained on balanced multilingual data – otherwise, the tokenizer will create a structural disadvantage for non-majority languages that no amount of model training can overcome. The following code compares fertility across languages using GPT-2's English-optimized tokenizer, making the disparity concrete:

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("gpt2")

# Compare fertility across languages (same semantic content)
texts = {
    "English":  "Artificial intelligence is transforming the world.",
    "German":   "Kuenstliche Intelligenz veraendert die Welt.",
    "Chinese":  "ren gong zhi neng zheng zai gai bian shi jie.",
    "Korean":   "ingoing jineungi segyereul byeonhwasikigo itda.",
}
for lang, text in texts.items():
    tokens = tokenizer.encode(text)
    words = text.split()
    fertility = len(tokens) / len(words)
    print(f"{lang:>10s}: {len(tokens):2d} tokens / {len(words)} words "
          f"= fertility {fertility:.2f}  |  "
          f"{tokenizer.convert_ids_to_tokens(tokens)}")
# English: ~1.2 fertility; other languages: 2-4x higher
```

Sidebar: The Tokenization Tax – Why Non-English Languages Pay More

When OpenAI released GPT-2 in 2019, its tokenizer had a vocabulary of 50,257 tokens trained predominantly on English web text. For English, the tokenizer was remarkably efficient: common words like "the," "is," and "running" were single tokens, and even uncommon words like "transformers" required only 2 tokens. For non-English languages, the picture was starkly different. The Chinese word for "artificial intelligence" required 6 tokens. Arabic text averaged 4.2 tokens per word. Hindi text averaged 5.1 tokens per word. This disparity has concrete financial consequences. OpenAI's API pricing is per token, so Chinese users pay 3–4 times more per semantic unit than English users. The context window (measured in tokens) holds 3–4 times fewer Chinese semantic units than English words, effectively giving non-English users a shorter working memory. Petrov et al. [2023] measured this systematically across 17 languages and found that the cost ratio ranged from 1.0 (English) to 15.0 (Myanmar). The BLOOM project [BigScience, 2022] was the first large-scale attempt to address this, training its tokenizer on a balanced

multilingual corpus covering 46 languages and achieving significantly more equitable fertility. The lesson: tokenizer design is not merely a technical detail. It is an equity issue that determines who benefits from language technology and who is left behind.

10.4.3 Toward Tokenizer-Free Models

Given the complications introduced by tokenization – the vocabulary size trade-off, the fertility problem, the language dependence of BPE merges – a natural question is whether we can eliminate the tokenizer entirely. Several recent research efforts have explored this direction, operating directly on raw bytes without any subword segmentation. The most prominent is ByT5 [Xue et al., 2022], a variant of T5 (Chapter 9) that takes UTF-8 byte sequences as input and produces byte sequences as output, with no tokenizer in the pipeline. ByT5 uses a modified architecture with a deeper encoder to compensate for the longer input sequences: where mT5 (multilingual T5) processes a sentence as perhaps 15 subword tokens, ByT5 processes the same sentence as 60–80 bytes. The results were encouraging but mixed. On English benchmarks, ByT5 matched mT5's performance at comparable compute. On morphologically complex languages and noisy text (misspellings, social media abbreviations), ByT5 was notably better, because it never had to decompose an unknown word into potentially meaningless subword fragments. On the other hand, the longer sequences increased training and inference time substantially, and for long documents the quadratic attention cost became prohibitive.

MegaByte [Yu et al., 2023] addresses the sequence length problem with a hierarchical architecture. Instead of feeding individual bytes through a single Transformer, MegaByte groups bytes into fixed-size patches (e.g., 8 bytes per patch) and processes patches with a "global" Transformer at reduced sequence length. Within each patch, a smaller "local" model predicts individual bytes conditioned on the global model's output. This patch-based approach reduces the effective sequence length by the patch size – an 8-byte patch turns 2,048 bytes into 256 patches for the global model – while maintaining byte-level output granularity. The approach is conceptually similar to how vision Transformers (ViT) process images as patches of pixels rather than individual pixels.

It would be premature to declare the tokenizer dead. As of current practice, every production large language model – GPT-4, Claude, Gemini, LLaMA, Mistral – uses subword tokenization. Byte-level models remain a research direction rather than a deployed standard. The computational overhead of longer sequences has not been fully overcome, and the marginal gains over well-designed multilingual tokenizers do not yet justify the cost for most applications. But the trajectory is clear: the trend is toward reducing the tokenizer's role, and the eventual elimination of this pipeline stage is a plausible – if not yet imminent – outcome. Whether that future arrives in two years or ten is genuinely uncertain, and anyone who claims to know the timeline is guessing.

10.5 Data Curation at Scale

A model can only learn patterns that exist in its training data. So where does the data come from, and who decides what "good" data looks like?

We have so far focused on how text is split into tokens. We now turn to the complementary question: what text should the model see in the first place? The shift in perspective here is dramatic. In the early days of neural NLP (2013–2017), researchers obsessed over architectures – LSTMs versus GRUs, attention mechanisms, residual connections – and treated training data as an afterthought.

You grabbed whatever corpus was available, tokenized it with standard tools, and fed it to the model. The first hint that data mattered as much as architecture came from GPT-2, whose WebText corpus was curated using a surprisingly simple filter: only web pages linked from Reddit posts with at least 3 upvotes. The curation was crude, but GPT-2's outputs were notably more coherent than those of models trained on unfiltered web crawls. The lesson crystallized with T5 [Raffel et al., 2020], which showed that the same Transformer architecture trained on C4 (a cleaned version of Common Crawl) dramatically outperformed the same architecture trained on raw Common Crawl. By 2023, the consensus had fully inverted: the model *is* the data. When Meta released LLaMA [Touvron et al., 2023], the paper's most scrutinized section was not the architecture (a standard Transformer) but the data mixture.

10.5.1 Data Sources: Web Crawls, Books, Code

Modern large language model training corpora are multi-source mixtures, typically dominated by web text but supplemented with curated sources chosen for specific contributions to model capability. The dominant source of raw text is Common Crawl, a non-profit organization that crawls the web monthly and makes its archives publicly available. Common Crawl contains over 250 billion web pages accumulated since 2008, representing petabytes of raw HTML. The data is massive but noisy: it contains spam, boilerplate navigation text, duplicate pages, low-quality machine-generated content, and material in hundreds of languages at widely varying quality levels. No responsible team feeds raw Common Crawl directly to a language model.

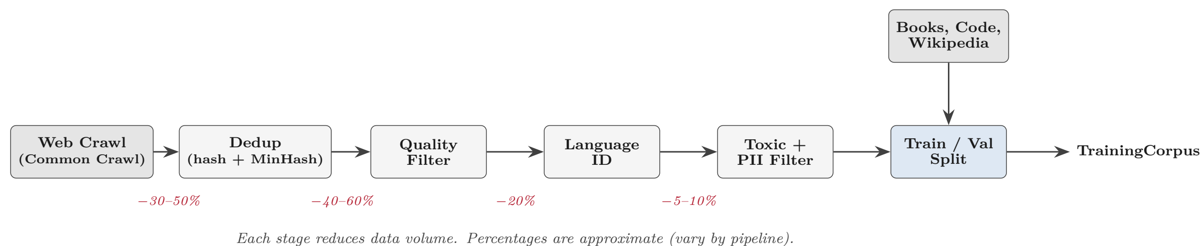

Figure 3: Figure 10.4 – Data Pipeline Flowchart

The curated corpora that have shaped the field include C4 (Colossal Clean Crawled Corpus), a 750GB dataset constructed by Raffel et al. [2020] from Common Crawl by applying heuristic quality filters (minimum document length, punctuation requirements, deduplication) and retaining only English text. C4 became the training corpus for T5 and established the template for subsequent data curation efforts. The Pile [Gao et al., 2021] took a different approach: rather than cleaning a single web source, it assembled 22 diverse sub-datasets including PubMed (biomedical literature), ArXiv (scientific preprints), GitHub (source code), Stack Exchange (technical Q&A), FreeLaw (legal documents), and others, each contributing specific capabilities. The Pile totals 825GB and was designed under the hypothesis that *diversity* of sources matters as much as *volume*. RedPajama [Together, 2023] was an open-source effort to replicate the training data composition described in the LLaMA paper, producing 1.2 trillion tokens from seven source categories. RefinedWeb [Penedo et al., 2023] represents the current state of the art in web-only data curation, applying aggressive deduplication and quality filtering to Common Crawl to produce 5 trillion tokens of English text that, remarkably, matched the quality of multi-source mixtures like The Pile when used to train the Falcon models.

The composition of the training mixture directly affects model capabilities in ways that are by now well documented. Increasing the proportion of source code improves the model's ability to reason about structured problems and generate code. Including scientific papers improves factual accuracy on technical topics. Including books and long-form prose improves the model's ability to maintain coherence over extended passages. The training corpus is, in a real sense, the model's curriculum, and its composition determines what the model can do.

10.5.2 Deduplication and Quality Filtering

Raw web crawls contain enormous amounts of duplication – both exact duplicates (identical pages crawled at different times or hosted on mirror sites) and near-duplicates (pages that differ only in headers, footers, or minor edits). Lee et al. [2022] found that even the supposedly "clean" C4 dataset contained 3.04% exact duplicates, and near-duplication rates were far higher. Duplicate training data is not merely wasteful; it is actively harmful. Models trained on duplicated data exhibit increased memorization of repeated content, reduced output diversity, and a higher tendency to regurgitate training text verbatim – a privacy and copyright concern. Lee et al. [2022] showed that training on a deduplicated corpus of X tokens outperformed training on a raw corpus of $2X$ tokens on downstream benchmarks while also reducing verbatim memorization by a factor of 10. Less data, when it is the right data, genuinely beats more data.

Figure 4: Figure 10.5 – Data Quality Filtering Examples

The deduplication pipeline operates in two stages. Exact deduplication uses cryptographic hashing (typically SHA-256 or MD5) to assign each document a unique fingerprint; documents with identical hashes are duplicates and all but one copy is removed. This stage is computationally cheap and removes the lowest-hanging fruit. Near-duplicate detection is more expensive and uses *MinHash with Locality-Sensitive Hashing* (LSH) to identify documents that share a high fraction of their content even if they are not character-for-character identical. The approach works by computing a set of hash signatures (MinHash values) for each document's n-gram shingles and then using LSH to efficiently find documents whose signature sets overlap above a threshold (typically 80% Jaccard similarity). We omit the algorithmic internals of MinHash here – the reader interested in the details should consult the original work by Broder [1997] – but the practical outcome is that near-duplicate detection can reduce a web corpus by 30–50% while preserving unique content.

Quality filtering aims to remove content that is syntactically or semantically low-quality: spam, boilerplate, machine-generated text, incoherent rambling, and pages consisting primarily of navigation menus or advertisements. Two main approaches dominate. The first is *classifier-based filtering*: train a binary classifier (typically a lightweight fastText model) on Wikipedia text as positive examples and random web text as negative examples, then retain only documents that the classifier scores above a threshold. The second is *perplexity-based filtering*: compute the perplexity of each document under a reference language model (typically a KenLM n-gram model trained on Wikipedia) and remove documents with perplexity above a threshold, under the assumption that high-perplexity text is incoherent or non-standard. Both approaches are imperfect – they can filter out legitimate but unusual text, such as poetry, dialect writing, or technical jargon – but the net effect on model quality is consistently positive. The C4 pipeline [Raffel et al., 2020] used heuristic rules (minimum document length, presence of terminal punctuation, filtering of "dirty words") and achieved good results with these simple interventions. More recent pipelines like RefinedWeb use combinations of all three approaches: heuristic rules, classifier-based scoring, and perplexity filtering.

10.5.3 Data Mixing and Domain Balancing

After deduplication and quality filtering, the final stage of data curation is *mixing*: combining data from different sources at specific proportions to produce the training corpus. This stage is deceptively simple in concept – just concatenate data from different sources – but the mixing proportions have a large and well-documented impact on model capabilities. If web text constitutes 90% of available data and source code constitutes 2%, training proportional to source size means code is heavily underrepresented. But code is disproportionately valuable for developing reasoning and structured generation capabilities. Similarly, scientific papers are a small fraction of total text but critical for factual accuracy, and Wikipedia is tiny by volume but dense with well-organized knowledge.

The standard solution is *temperature-scaled sampling*, where the probability of drawing a training example from source s is proportional to the source size raised to a power $1/T$: $P(s) \propto |s|^{1/T}$. With temperature $T = 1$, sampling is proportional to source size (large sources dominate). As T increases, the distribution flattens, upsampling small sources relative to their natural size. In the limit $T \to \infty$, all sources are sampled uniformly regardless of size. In practice, teams also use explicit proportions rather than a single temperature parameter. The LLaMA training mixture, for example, used 67% Common Crawl, 15% C4, 4.5% GitHub, 4.5% Wikipedia, 4.5% books, 2.5% ArXiv, and 2% Stack Exchange [Touvron et al., 2023]. These proportions were not derived from any theoretical formula but from empirical experiments comparing downstream performance across different mixtures.

A related concern is *data epochs* – how many times the model sees each training example. High-quality sources like Wikipedia and curated books are often repeated 2–5 times during training (multi-epoch training), while web text is typically seen only once. Repeating high-quality data does help, but there are diminishing and eventually negative returns: beyond approximately 4 epochs on the same data, models begin to overfit and memorize rather than generalize. The trade-off between data repetition and data volume is an active area of research, with recent work suggesting that the optimal number of repetitions depends on both the quality of the data and the total compute budget available.

The practical takeaway is that data mixing is one of the most impactful decisions in language model development, sitting alongside model architecture and training hyperparameters as a primary lever for performance. The analogy to nutrition is apt: a model's training data is its diet, and

the proportion of different food groups – high-quality prose, technical writing, code, multilingual text – determines what capabilities the model develops. Too much junk food (low-quality web text) produces a weak model; a balanced diet (diverse, high-quality sources at carefully chosen proportions) produces a capable one. Data curation is nutritional science for language models, and it is no coincidence that the teams building the strongest models invest as heavily in data pipelines as they do in GPU clusters.

We have now covered the complete pipeline that transforms raw text into the input a language model actually consumes: tokenization splits text into subword units that balance vocabulary coverage against sequence compactness, and data curation ensures that the patterns available for learning are diverse, high-quality, and representative. These are the unglamorous foundations – the plumbing and wiring – that determine whether the elegant architectures of Chapter 8 and the clever pre-training objectives of Chapter 9 produce a capable model or a mediocre one. But a question looms over this entire pipeline that we have not yet addressed: how *big* should the model be? How much data does it need? How much compute is required to train it? In Chapter 11, we will discover that the answers follow surprisingly precise mathematical laws – the scaling laws that govern how pre-training loss decreases as model size, dataset size, and compute budget increase. These laws have become the engineering foundation for planning billion-dollar training runs.

With the practical machinery of tokenization and data in place, we make a natural transition to Chapter 11, where we investigate the scaling laws that govern how model performance improves with size, data, and compute.

Exercises

Exercise 10.1 (Theory – Basic). Given the following corpus with word frequencies:

- "h u g ": 10
- "p u g ": 5
- "h u g s ": 7
- "b u g ": 3

Compute the first 4 BPE merges by hand. At each step, show: (a) the complete frequency table of adjacent pairs, (b) the selected merge, (c) the updated corpus representation, and (d) the new vocabulary. *Hint:* The most frequent pair in the first step will involve "u" and "g," since they appear adjacent in every word.

Exercise 10.2 (Theory – Intermediate). Compare the compression ratios of BPE and Unigram tokenizers for the same vocabulary size. Define compression ratio as (number of characters in original text) / (number of tokens after tokenization). Given that BPE optimizes for frequency (most common pairs merged first) and Unigram optimizes for likelihood (most useful tokens retained), argue which algorithm should achieve better compression *on average*. Under what conditions might the other algorithm win? Consider the case of morphologically rich languages (e.g., Turkish, Finnish) where meaningful morphemes may not be the most *frequent* character pairs.

Exercise 10.3 (Theory – Intermediate). For a 1-million-token English corpus, estimate the average number of tokens per document (assume 500-word documents) for BPE vocabulary sizes of 1K, 10K, 32K, 64K, and 128K. Use the empirical observation that English averages approximately 4.5

characters per word and that a vocabulary of size $|V|$ covers approximately $1 - C/|V|^{0.5}$ of the text as single tokens, where C is a corpus-specific constant (use $C = 200$ for English web text). At what vocabulary size do diminishing returns make further expansion unjustified? Relate your analysis to the vocabulary sizes chosen by GPT-2 (50K), LLaMA (32K), and GPT-4 (100K).

Exercise 10.4 (Theory – Basic). Compute the fertility of GPT-2's tokenizer on the following three sentences: (1) "The quick brown fox jumps over the lazy dog" (English), (2) "Der schnelle braune Fuchs springt ueber den faulen Hund" (German), (3) a sentence in a non-Latin script language of your choice. Report the fertility for each sentence, explain why the values differ, and discuss the downstream consequences for: (a) API cost, (b) effective context window use, and (c) inference latency. *You may use the HuggingFace tokenizer for computation.*

Exercise 10.5 (Programming – Basic). Implement the BPE algorithm from scratch in Python (no external tokenization libraries). Your implementation should: (a) accept a corpus as a dictionary mapping words (space-separated character sequences) to frequencies, (b) perform K merge operations, (c) output the merge table and final vocabulary, and (d) include an `encode` function that tokenizes new text using the learned merges. Train your BPE on a provided corpus of 1,000 sentences with $K = 500$ merges and report the average tokens per word on 10 held-out sentences.

Exercise 10.6 (Programming – Intermediate). Using the SentencePiece library, train both a BPE and a Unigram tokenizer on the same English corpus (use the first 10,000 sentences of WikiText-103) with a vocabulary size of 8,000. For each tokenizer: (a) tokenize 100 held-out sentences and compute the average tokens per sentence, (b) compute the vocabulary overlap between the two models (what fraction of tokens appear in both vocabularies?), and (c) compare the segmentations of five morphologically complex words: "unbelievable," "internationalization," "antidisestablishmentarianism," "preprocessing," and "counterrevolutionary." Which tokenizer produces more linguistically meaningful segmentations?

Exercise 10.7 (Programming – Intermediate). Measure fertility across 5 languages (English, German, Chinese, Arabic, Hindi) using two tokenizers: GPT-2's tokenizer and BLOOM's multilingual tokenizer. For each language, tokenize 50 sentences from the FLORES benchmark (or construct your own parallel sentences) and compute average fertility. Produce a grouped bar chart comparing the two tokenizers across all 5 languages. Which tokenizer is more equitable? Quantify the "tokenization tax" as the ratio of non-English fertility to English fertility for each tokenizer.

Exercise 10.8 (Programming – Advanced). Build a simple data quality filtering pipeline. Download a sample of 1,000 web pages from Common Crawl (or use a local web scrape). Implement three filtering stages: (1) language identification using the `langdetect` library (retain only English documents), (2) exact deduplication using SHA-256 hashing, (3) quality scoring using at least two heuristic rules (e.g., minimum document length of 200 characters, maximum repetition ratio below 0.3, presence of at least one sentence-ending punctuation mark per 100 characters). Report how many documents each stage removes, examine 5 documents that were filtered at each stage, and discuss whether the filtering decisions seem appropriate.

References

Gage, P. (1994). A New Algorithm for Data Compression. *The C Users Journal*, 12(2), 23–38.

Sennrich, R., Haddow, B., & Birch, A. (2016). Neural Machine Translation of Rare Words with

Subword Units. In *Proceedings of ACL*, 1715–1725.

Schuster, M. & Nakajima, K. (2012). Japanese and Korean Voice Search. In *Proceedings of ICASSP*, 5149–5152.

Kudo, T. & Richardson, J. (2018). SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing. In *Proceedings of EMNLP (System Demonstrations)*, 66–71.

Kudo, T. (2018). Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates. In *Proceedings of ACL*, 66–75.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language Models are Unsupervised Multitask Learners. OpenAI Technical Report.

Provilkov, I., Emelianenko, D., & Voita, E. (2020). BPE-Dropout: Simple and Effective Subword Regularization. In *Proceedings of ACL*, 1882–1892.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., et al. (2020). Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *JMLR*, 21(140), 1–67.

Gao, L., Biderman, S., Black, S., Golding, L., Hoppe, T., et al. (2021). The Pile: An 800GB Dataset of Diverse Text for Language Modeling. arXiv:2101.00027.

Xue, L., Barua, A., Constant, N., Al-Rfou, R., Narang, S., et al. (2022). ByT5: Towards a Token-Free Future with Pre-trained Byte-to-Byte Models. *TACL*, 10, 291–306.

Lee, K., Ippolito, D., Nystrom, A., Zhang, C., Eck, D., Callison-Burch, C., & Carlini, N. (2022). Deduplicating Training Data Makes Language Models Better. In *Proceedings of ACL*, 8424–8445.

BigScience Workshop. (2022). BLOOM: A 176B-Parameter Open-Access Multilingual Language Model. arXiv:2211.05100.

Petrov, A., La Malfa, E., Torr, P., & Biber, A. (2023). Language Model Tokenizers Introduce Unfairness Between Languages. In *Advances in NeurIPS*.

Together. (2023). RedPajama: An Open Dataset for Training Large Language Models. Technical Report.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., et al. (2023). LLaMA: Open and Efficient Foundation Language Models. arXiv:2302.13971.

Penedo, G., Malartic, Q., Hesslow, D., Cojocaru, R., Cappelli, A., et al. (2023). The RefinedWeb Dataset for Falcon LLM: Outperforming Curated Corpora with Web Data, and Web Data Only. In *Advances in NeurIPS*.

Yu, L., Simig, D., Flaherty, C., Aghajanyan, A., Zettlemoyer, L., & Lewis, M. (2023). MEGABYTE: Predicting Million-byte Sequences with Multiscale Transformers. In *Advances in NeurIPS*.

Broder, A. Z. (1997). On the Resemblance and Containment of Documents. In *Proceedings of the Compression and Complexity of Sequences*, 21–29.