

Objective vs. Search: Decomposing What Makes a Good Tokeniser

Ahmetcan Yavuz¹ Clara Meister² Tiago Pimentel¹

¹ETH Zürich, ²EPFL

ayavuz@ethz.ch, clara.meister@epfl.ch, tiago.pimentel@inf.ethz.ch

🔗 Ahmetcanyvz/comp-vs-like 🤖 Ahmetcanyvz/comp-vs-like

Abstract

Two dominant tokenisation algorithms are used by modern language models: byte-pair encoding (BPE) and UnigramLM. These differ along two orthogonal axes: their *optimisation objective* (compression vs. log-likelihood) and their *search procedure* (bottom-up merging vs. top-down pruning). Existing comparisons confound these axes, making it unclear whether their observed differences stem from *what* is being optimised vs. *how* it is being optimised. We disentangle the two by introducing two new tokenisation algorithms that complete this 2×2 design space: BottomUpLL, a bottom-up likelihood-based tokeniser, and TopDownComp, a top-down compression-based tokeniser. We train language models with tokenisers produced by each algorithm, varying: model size, vocabulary sizes, and domain (English-only vs. multilingual). Evaluating models on bits-per-byte, we find that the search procedure—not the objective—is the dominant factor: bottom-up tokenisers consistently achieve lower bits-per-byte in most settings. Evaluating models on the BLiMP task, however, shows no consistent relationship between design choice and performance. Overall, our results disentangle the effect of tokeniser design choices on language modelling performance, offering concrete guidance for their more principled construction.

1 Introduction

Before a language model (LM) processes text, its raw character string is mapped to a token string, which defines the model’s input. This mapping is performed by a **tokeniser**, a foundational component of modern language modelling pipelines, used in virtually all state-of-the-art models (Team, 2026; OpenAI et al., 2025; Team et al., 2026). A tokeniser is defined by several attributes. Some, such as the vocabulary size, are set by the practitioner while others, such as the vocabulary itself, are learned from data by a **tokeniser learning algorithm**. Two such algorithms account for most

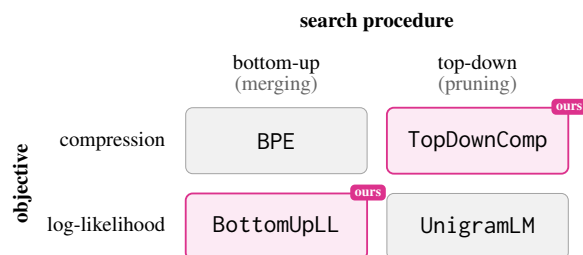


Figure 1: Design choice factorisation for tokenisation.

tokenisers in current use: (i) byte-pair encoding (BPE), which learns a vocabulary that maximises the compression of a training corpus (Gage, 1994; Sennrich et al., 2016); and (ii) UnigramLM, which learns a vocabulary that maximises the unigram log-likelihood of a training corpus (Kudo, 2018).

Despite their widespread usage, though, it remains unclear what makes a good tokeniser. BPE and UnigramLM are used as the default algorithms; yet their design choices have received little direct examination, and upon closer inspection, confound many of the existing comparisons. As a concrete example, the results of Schmidt et al. (2024)—that language models trained with BPE often perform better than those trained with UnigramLM—are commonly read as evidence that compression is a better **optimisation objective** than unigram log-likelihood. Optimisation objectives, though, are only one facet of a tokeniser learning algorithm.

BPE and UnigramLM also differ in the **search procedure** used to maximize the objective.¹ Given a dataset, BPE uses a bottom-up search, greedily merging symbols to maximise compression. UnigramLM relies on a top-down search, starting from an oversized vocabulary and iteratively pruning tokens to maximise log-likelihood. BPE and UnigramLM therefore differ along two axes at once:

¹The optimisation problem that such algorithms must solve is, in general, NP-hard (Kozma and Voderholzer, 2024; Whittington et al., 2025; Kastreva et al., 2026), so tokeniser learning algorithms typically approximate a solution with a heuristic search procedure.

the optimisation objective and the search procedure. A comparison between the two cannot attribute the difference in performance to either axis alone.

In this work, we disentangle these two axes. We introduce BottomUpLL, a bottom-up likelihood-based tokeniser that mirrors BPE’s merge procedure but optimises corpus log-likelihood instead of compression. We also introduce TopDownComp, a top-down compression-based tokeniser that follows UnigramLM’s pruning procedure but optimises compression. Together with BPE and UnigramLM, these methods form a 2×2 study crossing objective (compression vs. log-likelihood) and search procedure (bottom-up vs. top-down); see Fig. 1.

We train language models with tokenisers produced by all four algorithms, evaluating them in English-only and multilingual settings. We analyse these models with a combination of extrinsic and intrinsic evaluations. Notably, our results show that search procedure has a stronger influence on bits-per-byte (BPB) than the training objective, with bottom-up tokenisers achieving the best BPB scores in nearly all experimental conditions (BPE outperforms TopDownComp, and BottomUpLL outperforms UnigramLM). This ordering does not carry over to grammaticality judgements (on a BLiMP task), though, where neither the objective nor the search procedure separates the tokenisers consistently. Interestingly, at small vocabulary settings, however, the objective still matters, with likelihood-based tokenisers outperform compression-based ones. Overall, our results highlight the importance of both optimisation objective and search procedure to tokeniser training algorithm design.

2 Tokenisation

Language modelling starts from raw text, which can be represented as **character-strings** $\mathbf{c} \in \Sigma^*$; these are finite sequences of characters $\mathbf{c} = c_1 c_2 \dots c_{|\mathbf{c}|}$ over alphabet Σ .² A tokeniser’s job is then to *segment* these character-strings into **token-strings** $\mathbf{s} \in \mathcal{S}^*$: sequences $\mathbf{s} = \langle s_1, s_2, \dots, s_{|\mathbf{s}|} \rangle$, where each symbol s_t is called a token and represents a non-empty character span. Whenever $\mathbf{s}$ is a segmentation of $\mathbf{c}$, we say these strings are equivalent, which we denote as $\mathbf{c} \stackrel{\circ}{=} \mathbf{s}$, i.e.:³

$$\mathbf{c} \stackrel{\circ}{=} \mathbf{s} \iff \mathbf{c} = s_1 \circ s_2 \circ \dots \circ s_{|\mathbf{s}|}, \quad (1)$$

²As usual, Σ^* denotes the Kleene star of Σ , while Σ^+ denotes the set of all non-empty strings over Σ .

³We restrict our discussion to *lossless* tokenisers here.

Notably, tokenisers are usually not allowed to output any possible segmentation of $\mathbf{c}$, being instead constrained to segment it using only the tokens in a finite set called its **vocabulary** $\mathcal{S} \subset \Sigma^+$. To ensure that any character-string can be represented as tokens, we enforce $\Sigma \subseteq \mathcal{S}$.

How do we convert between character- and token-strings, though? Given just a vocabulary $\mathcal{S}$, there are multiple different ways a character-string could be mapped to items in that vocabulary. For example, if $\mathcal{S} = \{a, aa, aaa\}$, then the character-string $\mathbf{c} = aaa$ may be encoded as either $\langle a, aa \rangle$, $\langle aa, a \rangle$, or $\langle a, a, a \rangle$, which we refer to as **segmentations**. Consequently, a tokeniser is defined not only by its vocabulary, but also by *how* it maps character-strings to segmentations. This mapping is the job of an **encoding function**, $\text{tok} : \Sigma^* \rightarrow \mathcal{S}^*$, which segments the original character-string into tokens; by definition, we have thus that $\mathbf{c} \stackrel{\circ}{=} \text{tok}(\mathbf{c})$. Finally, a tokeniser also contains a **decoding function**, $\text{detok} : \mathcal{S}^* \rightarrow \Sigma^*$, which converts token-strings back into characters, being typically defined as the concatenation of each token’s characters: $\text{detok}(\mathbf{s}) \stackrel{\text{def}}{=} s_1 \circ s_2 \circ \dots \circ s_{|\mathbf{s}|}$. Formally, we thus define a tokeniser as the tuple $\mathbb{T} \stackrel{\text{def}}{=} \langle \mathcal{S}, \text{detok}, \text{tok} \rangle$.

3 Learning a Tokeniser: Optimisation Objective vs. Search Procedure

How do we select such a tokeniser? Here, we cast the learning of a tokeniser as the combination of two design components: an objective function and a search procedure.

3.1 Objective Functions

Given a tokeniser $\mathbb{T}$ and a dataset $\mathcal{D} = \{\mathbf{c}_m\}_{m=1}^M$, an objective function $\mathcal{G}$ assigns a score to the tokeniser. Different objectives encode different notions of what makes a tokeniser useful. We focus on two standard objectives used by tokeniser learning algorithms: compression and log-likelihood.

Under the **compression** objective, the quantity to be minimised is the total number of tokens that the tokeniser produces on the dataset:

$$\mathcal{G}_{\text{comp}}(\text{tok}, \mathcal{D}) \stackrel{\text{def}}{=} \sum_{\mathbf{c} \in \mathcal{D}} |\text{tok}(\mathbf{c})|, \quad (2)$$

This is appealing because shorter token strings improve effective context usage and reduce the number of model steps needed to process a dataset.

Under the **log-likelihood** objective, the minimised quantity is the negative log-probability that

a given probabilistic model p_θ assigns the corpus:

$$\mathfrak{G}_\Pi(\text{tok}, \mathcal{D}) \stackrel{\text{def}}{=} - \sum_{\mathbf{c} \in \mathcal{D}} \log p_\theta(\text{tok}(\mathbf{c})). \quad (3)$$

where p_θ 's parameters θ are themselves optimised to model this tokeniser's text. This objective is likewise appealing, since log-likelihood is also the training objective of the language models for which these tokenisers are designed—so a log-likelihood optimal tokeniser would, by construction, lead to better models. Optimising log-likelihood directly, however, is intractable, as this objective requires fully training language models p_θ for its evaluation.

In practice, to make this objective tractable, we typically significantly restrict the class of models p_θ . Here, we follow UnigramLM in restricting p_θ to unigram language models. For a fixed encoding function tok , let $n_s^\mathcal{D}$ denote the count of token s in the tokenised corpus, and let $N^\mathcal{D} \stackrel{\text{def}}{=} \sum_{s \in \Sigma} n_s^\mathcal{D}$ be the total number of tokens. We then define p_θ as

$$p_\theta(\mathbf{s}) = \prod_{t=1}^{|\mathbf{s}|} p_\theta(s_t), \quad p_\theta(s) = \frac{n_s^\mathcal{D}}{N^\mathcal{D}}. \quad (4)$$

Concretely, we will thus use **unigram log-likelihood** as the objective here, as opposed to 'full' log-likelihood, and, for the rest of this paper, log-likelihood stands for unigram log-likelihood.

3.2 Search Procedures

Given one of these objectives and a fixed vocabulary budget K , we might expect a tokeniser to be chosen by directly optimising

$$\text{tok}_{\text{opt}} = \underset{\text{tok} \in \mathcal{T}_K}{\text{argmin}} \mathfrak{G}(\text{tok}, \mathcal{D}), \quad (5)$$

where $\mathcal{T}_K$ denotes whichever class of tokenisers is under consideration. Unfortunately, this optimisation problem is computationally intractable; the compression variant is provably NP-hard (Kozma and Voderholzer, 2024; Whittington et al., 2025; Kastreva et al., 2026), and we suspect the log-likelihood variant to be as well. In practice, heuristic **search procedures** are therefore required. Such procedures specify both a restricted class of tokenisers and a strategy for selecting an element from that class. We focus on two standard search procedures: bottom-up construction and top-down pruning.

The **bottom-up** procedure builds a tokeniser from the bottom up: the vocabulary starts as the alphabet Σ and grows one token at a time through *merge operations*. Given a merge $m = \langle s', s'' \rangle$,

applying it to a token-string rewrites every (non-overlapping) occurrence of bigram s', s'' into the token $s^{\text{new}} = s' \circ s''$. A merge list $\mathbf{m} = m_1, \dots, m_k$ then defines an encoding function $\text{tok}_\uparrow[\mathbf{m}]$: starting from $\mathbf{c}$ as a token-string over Σ (one token per character), it applies $m_1, \dots, m_k$ in order. The final token-string is our tokenised text. We write the tokeniser class defined by this procedure as $\mathcal{T}_K^\uparrow = \{\text{tok}_\uparrow[\mathbf{m}] \mid \mathbf{m} \in (\Sigma^+ \times \Sigma^+)^K\}$. To select an element from this class, we then start from an empty merge list, and greedily add one merge per step to it, locally optimising an objective $\mathfrak{G}$:

$$m^* = \underset{m \in \Sigma^+ \times \Sigma^+}{\text{argmin}} \mathfrak{G}(\text{tok}_\uparrow[\mathbf{m}_{<k}, m], \mathcal{D}), \quad (6)$$

where $\text{tok}_\uparrow[\mathbf{m}_{<k}, m]$ denotes the encoding function obtained by applying the current merge list $\mathbf{m}_{<k}$ followed by the candidate m . The selected merge m^* is then appended to the merge list. We refer to a candidate merge's effect on the objective as its **merge gain**, and write it $\Delta_{s^1, s^2}^{\mathfrak{G}(\cdot, \mathcal{D})}$.

By contrast, the **top-down** procedure starts from a large candidate vocabulary $\mathcal{S}_0$, with $|\mathcal{S}_0| \gg K$, and removes tokens until $|\mathcal{S}| = |\Sigma| + K$. Notably, this procedure does not rely on merge lists, but instead commits to an **objective-optimal** encoding function $\text{tok}_\Rightarrow[\mathcal{S}]$, determined by the vocabulary alone. Under compression, $\text{tok}_\Rightarrow[\mathcal{S}](\mathbf{c})$ is the shortest valid segmentation of $\mathbf{c}$ using tokens from $\mathcal{S}$. Under log-likelihood, it is instead the most probable segmentation of $\mathbf{c}$ under p_θ .⁴ Top-down's tokeniser class can then be written as: $\mathcal{T}_K^\downarrow = \{\text{tok}_\Rightarrow[\mathcal{S}] \mid \mathcal{S} \subset \Sigma^+, \Sigma \subseteq \mathcal{S}, |\mathcal{S}| = |\Sigma| + K\}$. To select an item in this class, we then start from a large vocabulary $\mathcal{S}_0$, from which we iteratively prune the least critical tokens, quantified as the tokens whose removal has the least negative impact on the objective we're optimizing for:

$$s_\mathbf{x} = \underset{s \in \mathcal{S}_{k-1} \setminus \Sigma}{\text{argmin}} \mathfrak{G}(\text{tok}_\Rightarrow[\mathcal{S}_{k-1} \setminus \{s\}], \mathcal{D}). \quad (7)$$

Here, we index pruning steps by k , mirroring the bottom-up case. This yields $\mathcal{S}_k = \mathcal{S}_{k-1} \setminus \{s_\mathbf{x}\}$. Importantly, to ensure every string remains encodable, alphabet tokens are never removed; pruning then continues until $|\mathcal{S}| = |\Sigma| + K$.⁵ Analogously, we refer to a candidate deletion's effect on the objective as its **deletion cost**, written $\Delta_s^{\mathfrak{G}(\cdot, \mathcal{D})}$.

⁴Given a fixed p_θ , both cases can be computed efficiently. For log-likelihood, however, the parameters of p_θ itself depend on tok , creating a circular definition. In practice, the two are estimated jointly as we explain in §4.2.

⁵We note that, for efficiency reasons, in practice, multiple tokens are often pruned in batches.

4 Analysed Tokenisers: Old and New

Combining the objectives and search procedures above, we get a 2×2 design space for tokenisation algorithms (illustrated in Fig. 1). Existing tokeniser learning algorithms, namely BPE and UnigramLM, cover only two of these cells. Here, we introduce algorithms that instantiate the two other cells: BottomUpLL, a bottom-up likelihood-based method, and TopDownComp, a top-down compression-based method.⁶ Before any of these methods can be run, however, a loose end remains.

In the previous section, we describe the search procedures as greedy: at each step, it picks the merge m^* or deletion s_x which (locally) optimises an objective $\mathfrak{G}(\cdot, \mathcal{D})$. It did not specify, however, how this selection is performed. Taken literally, computing the argmin operations in Eqs. (6) and (7) would require running the objective function $\mathfrak{G}$ once per candidate to find the optimal one—a computationally impractical operation. Rather than recomputing $\mathfrak{G}$ every time, most tokenisation algorithms work with *incremental scores* instead:

$$m^* = \underset{\langle s^1, s^2 \rangle \in \Sigma^+ \times \Sigma^+}{\operatorname{argmax}} \Delta_{s^1, s^2}^{\mathfrak{G}(\cdot, \mathcal{D})} \quad (8a)$$

$$s_x = \underset{s \in \mathcal{S}_{k-1} \setminus \Sigma}{\operatorname{argmin}} \Delta_s^{\mathfrak{G}(\cdot, \mathcal{D})} \quad (8b)$$

Notably, these equations are equivalent to §3.2’s, as the pre-update objective values (e.g., $\mathfrak{G}(\text{tok}_\uparrow[m_{<k}], \mathcal{D})$) do not depend on the candidates, and are thus constant. Each search procedure then derives a way to compute (or approximate) these values efficiently, as we show next.

4.1 Bottom-up tokenisers: BPE and BottomUpLL

For a bottom-up method, each candidate is scored according to its merge gain $\Delta_{s^1, s^2}^{\mathfrak{G}(\cdot, \mathcal{D})}$. We now show how this score can be computed efficiently for the two objectives, starting with compression.

Lemma 1. *The change in compression due to merging a token-pair can be computed as:*

$$\Delta_{s^1, s^2}^{\mathfrak{G}_{\text{comp}}(\cdot, \mathcal{D})} = n_{s^1, s^2}^{\mathcal{D}} \quad (9)$$

where $n_{s^1, s^2}^{\mathcal{D}}$ denotes the number of non-overlapping $\langle s^1, s^2 \rangle$ sequences in $\mathcal{D}$.

⁶Our new methods are closely related to, e.g., WordPiece or PathPiece. We discuss this prior work in detail in §5.

Proof sketch. The full proof is in §A. Replacing an occurrence of the token-pair $m = \langle s^1, s^2 \rangle$ with the merged token $s^1 \circ s^2$ saves one token. Further, the number of non-overlapping occurrences of a token-pair is its number of possible merges. Merging the pair therefore shortens the corpus by exactly $n_{s^1, s^2}^{\mathcal{D}}$ tokens, which is its gain under $\mathfrak{G}_{\text{comp}}$. $\square$

Lemma 1 gives the standard BPE merge rule: the compression-optimal merge is the most (non-overlappingly) frequent token-pair. In practice, the token-pair counts $n_{s^1, s^2}^{\mathcal{D}}$ are computed once and then updated incrementally after each merge, rather than recomputed from scratch (see §B), which allows us to run BPE efficiently. We can derive an analogous equivalence for BottomUpLL.

Lemma 2. *The change in log-likelihood due to merging a token-pair can be computed as:⁷*

$$\begin{aligned} \Delta_{s^1, s^2}^{\mathfrak{G}_{\text{ll}}(\cdot, \mathcal{D})} = & (n_{s^1}^{\mathcal{D}} - n_{s^1, s^2}^{\mathcal{D}}) \log(n_{s^1}^{\mathcal{D}} - n_{s^1, s^2}^{\mathcal{D}}) - n_{s^1}^{\mathcal{D}} \log n_{s^1}^{\mathcal{D}} \\ & + (n_{s^2}^{\mathcal{D}} - n_{s^1, s^2}^{\mathcal{D}}) \log(n_{s^2}^{\mathcal{D}} - n_{s^1, s^2}^{\mathcal{D}}) - n_{s^2}^{\mathcal{D}} \log n_{s^2}^{\mathcal{D}} \\ & + n_{s^1, s^2}^{\mathcal{D}} \log n_{s^1, s^2}^{\mathcal{D}} \\ & - (N^{\mathcal{D}} - n_{s^1, s^2}^{\mathcal{D}}) \log(N^{\mathcal{D}} - n_{s^1, s^2}^{\mathcal{D}}) + N^{\mathcal{D}} \log N^{\mathcal{D}} \end{aligned} \quad (10)$$

Proof sketch. The full proof is in §C. Note that, if a token changes from count n_1 to count n_2 , its contribution to the log-likelihood objective changes by: $n_2 \log n_2 - n_1 \log n_1$. The lemma then follows trivially, noting that: (i) the old tokens s^1 and s^2 change from, e.g., frequency $n_{s^1}^{\mathcal{D}}$ to $(n_{s^1}^{\mathcal{D}} - n_{s^1, s^2}^{\mathcal{D}})$; (ii) the new token $s^1 \circ s^2$ changes from frequency 0 to $n_{s^1, s^2}^{\mathcal{D}}$. Further, the total count of tokens serves as a normalisation factor in all tokens, and thus enters the log-likelihood with the opposite sign; its contribution changes from $N^{\mathcal{D}}$ to $N^{\mathcal{D}} - n_{s^1, s^2}^{\mathcal{D}}$. $\square$

BottomUpLL can thus also be implemented efficiently, maintaining local count statistics and updating only affected candidate scores after each merge. Interestingly, an analysis of the merge gain definition in Lemma 2 shows that, for a token-pair to be selected: (i) the pair should occur often enough to matter, but (ii) it should also be favoured when its tokens co-occur more systematically than expected from their individual frequencies.

⁷For the case $s^1 = s^2$, the same calculation applies albeit with a modified count update: each merge consumes two occurrences of the same token s^1 and thus its count is reduced by $2n_{s^1, s^1}^{\mathcal{D}}$ instead; other parts of the update remain unchanged.

4.2 Top-down tokenisers: UnigramLM and TopDownComp

For top-down methods, each candidate’s score is given by the deletion cost $\Delta_s^{\mathfrak{G}(\cdot, \mathcal{D})}$. Unfortunately, this cost $\Delta_s^{\mathfrak{G}(\cdot, \mathcal{D})}$ is non-trivial to compute. This is due to the nature of the encoding function $\text{tok}_{\Rightarrow}[\mathcal{S}]$ used by top-down algorithms. Unlike in the bottom-up case, deleting a token here can change the preferred segmentation of an entire character-string, not only the local decomposition of the deleted token.⁸ For the log-likelihood objective, the history gets worse. The segmentations produced by $\text{tok}_{\Rightarrow}[\mathcal{S}]$ depend on p_{θ} , which is itself estimated from the token counts. Deleting a token changes those counts, which changes p_{θ} , which in turn changes how strings are segmented, which again changes the counts. Computing a deletion’s exact cost would thus require running this loop to convergence for each candidate token.

To approximate $\Delta_s^{\mathfrak{G}(\cdot, \mathcal{D})}$, we thus rely on a **local replacement approximation**: when scoring the deletion of s , we keep the rest of the current model fixed and estimate the loss by only directly replacing occurrences of s with an alternative segmentation under $\mathcal{S} \setminus \{s\}$.⁹ For log-likelihood, this then yields the following result.

Lemma 3. *Under a local replacement approximation, the estimated increase in the negative log-likelihood objective due to deleting a token $s \in \mathcal{S} \setminus \Sigma$ can be approximated as:*¹⁰

$$\Delta_s^{\mathfrak{G}(\cdot, \mathcal{D})} \approx n_s^{\mathcal{D}} (\log p_{\theta}(s) - \log p_{\theta}(\mathbf{s}_s^{\text{rep}})), \quad (11)$$

where p_{θ} is the unigram model at step $k - 1$, and $\mathbf{s}_s^{\text{rep}}$ is s ’s log-probability-optimal replacement segmentation, i.e., $\mathbf{s}_s^{\text{rep}} \stackrel{\text{def}}{=} \text{tok}_{\Rightarrow}[\mathcal{S}_{k-1} \setminus \{s\}](s)$.

Proof sketch. The full proof is in §D. Before deletion, each expected occurrence of s contributes

⁸For example, under a compression objective, suppose $\mathcal{S} = \Sigma \cup \{abc, def, cdef\}$. Then, string $abcdef$ is segmented as $\langle abc, def \rangle$. If abc is deleted, however, the shortest segmentation under the reduced vocabulary becomes $\langle a, b, cdef \rangle$, which also changes the previously used token def .

⁹Note that this introduces an asymmetry between our two search procedures: the bottom-up merge gains in the previous section are exact, whereas the top-down deletion costs are only approximate. We emphasise that this asymmetry is not a design choice on our part, but an inherent property of top-down search procedures, being itself present in UnigramLM. We return to its implications in our Limitations section.

¹⁰In practice, when computing this update, UnigramLM keeps segmentations latent and leverages p_{θ} to compute $n_s^{\mathcal{D}}$ as the expected count of s , instead of its exact count under $\text{tok}_{\Rightarrow}$.

$-\log p_{\theta}(s)$ to the negative log-likelihood. Under the local replacement approximation, deleting s replaces it with $\mathbf{s}_s^{\text{rep}}$, which contributes $-\log p_{\theta}(\mathbf{s}_s^{\text{rep}})$. As the number of such occurrences is $n_s^{\mathcal{D}}$, we get Eq. (11). $\square$

After each deletion, we update our encoding function $\text{tok}_{\Rightarrow}[\mathcal{S}_k]$ by re-estimating p_{θ} . We do this by updating θ via an expectation–maximisation procedure, maximising the log-likelihood of $\mathcal{D}$.¹¹ This updated p_{θ} defines the segmentations produced by $\text{tok}_{\Rightarrow}$, which we use to re-segment our dataset. We now present an analogous update for compression.

Lemma 4. *Under a local replacement approximation, the estimated change in compression due to deleting a token $s \in \mathcal{S} \setminus \Sigma$ can be approximated as:*

$$\Delta_s^{\mathfrak{G}_{\text{comp}}(\cdot, \mathcal{D})} = n_s^{\mathcal{D}} (|\mathbf{s}_s^{\text{rep}}| - 1), \quad (12)$$

where $\mathbf{s}_s^{\text{rep}}$ is the compression-optimal replacement of s after deletion: $\mathbf{s}_s^{\text{rep}} \stackrel{\text{def}}{=} \text{tok}_{\Rightarrow}[\mathcal{S}_{k-1} \setminus \{s\}](s)$.

Proof sketch. The full proof is in §E. After deleting s , the local replacement uses $|\mathbf{s}_s^{\text{rep}}|$ tokens instead, thus increasing the corpus length by $|\mathbf{s}_s^{\text{rep}}| - 1$. There are $n_s^{\mathcal{D}}$ such occurrences. $\square$

Updating this compression-based tokeniser after a deletion is easier than for log-likelihood. No parameters need to be estimated: the vocabulary $\mathcal{S}_k$ alone determines the encoding function, so we simply re-segment the corpus under $\text{tok}_{\Rightarrow}[\mathcal{S}_k]$.

5 Connection to Prior Work

As mentioned above, the tokenisation learning algorithms we propose are not entirely without precedent, and related to prior methods like WordPiece and PathPiece. In this section, we discuss this connections in detail.

5.1 BottomUpLL and Likelihood-Guided Merging

As noted in §4.1, BottomUpLL favours pairs of tokens that are not only frequent, but which co-occur more often than chance. In fact, let the pointwise mutual information between two tokens be:

$$\text{PMI}_{\text{adj}}(s^1, s^2) \stackrel{\text{def}}{=} \log \frac{p_{\theta}(s^1, s^2)}{p_{\theta}(s^1) p_{\theta}(s^2)} \quad (13a)$$

$$= \log \frac{N^{\mathcal{D}} n_{s^1, s^2}^{\mathcal{D}}}{n_{s^1}^{\mathcal{D}} n_{s^2}^{\mathcal{D}}}. \quad (13b)$$

¹¹The details of this procedure fall out of our work’s scope; for an accessible derivation, see Meister (2026).

Relying on the PMI to operationalise this notion of more-often-than-chance co-occurrence, we can use a Taylor approximation of $\Delta_{s^1, s^2}^{\mathcal{G}_{ll}(\cdot, \mathcal{D})}$ to make this explanation of BottomUpLL explicit.

Lemma 5. *Under a first-order Taylor approximation, the change in log-likelihood due to merging a token-pair can be approximated as:*

$$\Delta_{s^1, s^2}^{\mathcal{G}_{ll}(\cdot, \mathcal{D})} \approx n_{s^1, s^2}^{\mathcal{D}} (\text{PMI}_{\text{adj}}(s^1, s^2) - 1). \quad (14)$$

Proof sketch. The full proof is in §F. Starting from Lemma 2, apply a first-order Taylor expansion independently to each $x \log x$ term. Collecting the resulting terms returns the equation above. $\square$

Eq. (14) is closely related to the objective of another famous bottom-up tokeniser, **WordPiece**. WordPiece’s definition, however, is far from unified and has shifted over time. As originally proposed by Schuster and Nakajima (2012), WordPiece selects the unit that most increases the likelihood of the data—i.e., using the same objective as BottomUpLL—but approximates the local objective, selecting multiple merges in parallel and only approximately updating their model p_{θ} . WordPiece was later popularised by Wu et al. (2016), whose description suggests, in two consecutive sentences, optimising either a log-likelihood or a compression objective.¹² Accordingly, the widely used HuggingFace implementation of WordPiece optimises for compression (Wolf et al., 2020). Finally, WordPiece is often described as merging the pair with the highest PMI_{adj} (Schmidt et al., 2024; Lesci et al., 2025; Hugging Face, 2026), which relates to Eq. (14) but drops the $n_{s^1, s^2}^{\mathcal{D}}$ factor. BottomUpLL is thus closest to the original WordPiece’s description, but differs from all existing versions of this method.

Importantly, even closer to Eq. (14) is S-BPE (Vilar and Federico, 2021), which scores merges by a count-weighted PMI criterion. The two expressions are not identical, as S-BPE omits the -1 term, but are otherwise equivalent. We experiment with a tokenisation learning algorithm based on Lemma 5 in some of our experiments, which we then label as BottomUpLL $\approx$; when no such label is present, BottomUpLL refers to the method described by Lemma 2.

¹²“The wordpiece model is generated using a data-driven approach to maximize the language-model likelihood of the training data [...] Given a training corpus and a number of desired tokens D , the optimization problem is to select D wordpieces such that the resulting corpus is minimal in the number of wordpieces when segmented according to the chosen wordpiece model.” (Wu et al., 2016)

5.2 TopDownComp and Top-Down Compression

The closest prior method to TopDownComp is PathPiece (Schmidt et al., 2024), which likewise prunes a large initial vocabulary under a compression objective. The two differ in the deletion score: when scoring the removal of a token s , PathPiece also considers re-segmentation of other vocabulary tokens that contain s as a substring, whereas TopDownComp considers only s ’s own local replacement. Adopting the PathPiece variant would change the algorithm in more ways than just the objective, so we keep TopDownComp as the direct compression counterpart of UnigramLM, differing from it in objective alone.

6 Experimental Setup

We will now compare our four tokenisers: BPE, BottomUpLL, TopDownComp, and UnigramLM. All four are trained with an identical preprocessing pipeline (NFC normalisation followed by a byte-level pretokeniser using the GPT-2 regex) and on the same corpus, so that any difference between them stems only from the objective and the search procedure. §G gives the exact configuration under which we train our tokenisers. Full language model hyperparameter and architectural details are in §H.

English setup. For English experiments, we train our tokenisers on a subset with approximately 2B tokens from FineWeb-Edu (Lozhkov et al., 2024), using vocabulary sizes of 8k, 32k, and 128k. We then again use FineWeb-Edu to train language models using these tokenisers, evaluating 100M, 300M, 500M, and 1B parameter models.¹³ Results for 100M-parameter models are averaged over three random seeds, while larger models use one run per configuration. Finally, unless otherwise stated, models are trained with a Chinchilla-style token budget of approximately $20\times$ as many training tokens as model parameters.

Multilingual setup. Our multilingual corpus contains 20B tokens across five languages: 10B English tokens from FineWeb-Edu (Lozhkov et al., 2024), and 2.5B tokens each of German, Spanish, Turkish, and Chinese from FineWeb2 (Penedo et al., 2025). We train our tokenisers on a 10% subsample of this corpus, using a single vocabulary size of 128k. We then train 1B-parameter language models on the full corpus.

¹³Full hyperparameter and architectural details are in §H.

Vocab	Tokeniser	Objective	Search	$\mathfrak{G}_{\text{comp}} \downarrow$	$\mathfrak{G}_{\text{ll}} \downarrow$	Entropy	Zipf α	BPT $\uparrow$	Tok Len	Vocab Util $\uparrow$	Cov. 50%
8k	BPE	$\mathfrak{G}_{\text{comp}}$	bottom-up	59,559,835	6.259×10^8	10.508	1.033	3.844	5.53	99.9%	242
	TopDownComp	$\mathfrak{G}_{\text{comp}}$	top-down	59,304,113	6.259×10^8	10.554	0.919	3.861	5.67	99.3%	268
	BottomUpLL	$\mathfrak{G}_{\text{ll}}$	bottom-up	61,170,714	6.167×10^8	10.081	1.315	3.743	6.05	99.4%	152
	UnigramLM	$\mathfrak{G}_{\text{ll}}$	top-down	68,444,350	6.221×10^8	9.089	1.093	3.345	6.72	99.3%	39
32k	BPE	$\mathfrak{G}_{\text{comp}}$	bottom-up	50,327,096	5.544×10^8	11.016	1.195	4.549	6.52	99.8%	269
	TopDownComp	$\mathfrak{G}_{\text{comp}}$	top-down	52,204,230	5.667×10^8	10.856	1.074	4.386	6.29	99.8%	192
	BottomUpLL	$\mathfrak{G}_{\text{ll}}$	bottom-up	50,920,853	5.505×10^8	10.811	1.378	4.496	6.98	99.0%	211
	UnigramLM	$\mathfrak{G}_{\text{ll}}$	top-down	58,161,017	5.697×10^8	9.796	1.272	3.937	7.03	99.8%	51
128k	BPE	$\mathfrak{G}_{\text{comp}}$	bottom-up	46,859,565	5.216×10^8	11.132	1.484	4.886	7.06	98.9%	204
	TopDownComp	$\mathfrak{G}_{\text{comp}}$	top-down	50,312,483	5.484×10^8	10.900	1.443	4.551	6.01	99.8%	149
	BottomUpLL	$\mathfrak{G}_{\text{ll}}$	bottom-up	47,111,733	5.203×10^8	11.044	1.574	4.860	7.48	96.8%	191
	UnigramLM	$\mathfrak{G}_{\text{ll}}$	top-down	55,546,423	5.552×10^8	9.996	1.717	4.122	6.46	97.1%	53

Table 1: Intrinsic tokenisation results on a held-out test set (47,384 documents). Columns are defined in the Evaluation paragraph of §6. Tab. 5 in §J reports the multilingual counterpart of this table.

Evaluation. We report both extrinsic and intrinsic metrics. For *extrinsic* metrics, we report our language models’ bits per byte (BPB), computed on a held-out validation set, and minimal-pair grammatical accuracy, computed on BLiMP (Warstadt et al., 2020), MultiBLiMP (Jumelet et al., 2026), and ZhoBLiMP (Liu et al., 2026). For *intrinsic* metrics, we report $\mathfrak{G}_{\text{comp}}$ and $\mathfrak{G}_{\text{ll}}$. We also report the unigram entropy of our tokenised corpus, its bytes per token (BPT), average vocab token len (Tok Len), vocabulary utilisation (Vocab Util), Zipf α , and Coverage 50%. Detailed definitions are in §I.

7 Results

7.1 Intrinsic Evaluation

Compression vs. Log-likelihood. Tab. 1 presents both the $\mathfrak{G}_{\text{comp}}$ and $\mathfrak{G}_{\text{ll}}$ achieved by our tokenisers. From this table, we can see that, for a small vocabulary (of 8k tokens), the compression and log-likelihood scores achieved by the different tokenisers seem to align with their objective function: BPE and TopDownComp achieve better compression, while UnigramLM and BottomUpLL achieve better log-likelihood. For larger vocabularies, however, this surprisingly does not hold, with the bottom-up tokenisers consistently achieving both better compression and log-likelihood, irrespective of their objective function. Another result of interest is with respect to a tokeniser’s entropy, which, intuitively, measures how evenly a tokeniser spreads the frequency of its tokens. This quantity has a strong relationship to our optimisation objectives, which can be seen when we rewrite it:

$$\text{Ent}(\text{tok}, \mathcal{D}) \stackrel{\text{def}}{=} - \sum_{s \in \mathcal{S}} p_{\theta}(s) \log p_{\theta}(s) \quad (15a)$$

$$= \frac{\mathfrak{G}_{\text{ll}}(\text{tok}, \mathcal{D})}{\mathfrak{G}_{\text{comp}}(\text{tok}, \mathcal{D})} \quad (15b)$$

This follows directly from the definitions: writing $p_{\theta}(s) = n_s^{\mathcal{D}}/N^{\mathcal{D}}$, and noting that $N^{\mathcal{D}}$, the total number of tokens, is exactly $\mathfrak{G}_{\text{comp}}(\text{tok}, \mathcal{D})$, we get $-\sum_s p_{\theta}(s) \log p_{\theta}(s) = -\frac{1}{N^{\mathcal{D}}} \sum_s n_s^{\mathcal{D}} \log p_{\theta}(s) = \mathfrak{G}_{\text{ll}}(\text{tok}, \mathcal{D})/N^{\mathcal{D}}$. Looking at the entropy values in Tab. 1 we see a similar behaviour as before: for small vocabularies, entropy results align with the choice of tokenisation objective; for a larger vocabulary, however, top-down methods achieve consistently lower entropy. Future work should further investigate this relationship between objective and search procedure.

Distributional Properties of Tokens. Tab. 1 also reports our tokenisers’ Zipf α and Coverage 50%, where Zipf α is the negated slope of a frequency vs. rank log-log curve, and Coverage 50% is the smallest ℓ such that the ℓ most frequent tokens together account for at least half of all token occurrences in our corpus. A larger Zipf α thus means that token frequency decreases more rapidly with rank, while lower Coverage 50% values indicate that token occurrences are concentrated among fewer token types. Fig. 2 shows corresponding frequency–rank curves. Within each search family, the likelihood-based method induces a more concentrated token-frequency distribution than the compression-based method: BottomUpLL has higher Zipf α , lower entropy, and lower Coverage 50% than BPE, and UnigramLM shows the same pattern relative to TopDownComp. Thus, while the search procedure may dictate the values of $\mathfrak{G}_{\text{comp}}$ and $\mathfrak{G}_{\text{ll}}$, the used objective is still visible in a tokenised corpus’ empirical token-frequency distribution.

Other Intrinsic Metrics. Finally, Tab. 1 also presents our tokenisers’ vocabulary utilisation—i.e., the proportion of vocabulary items used at least once on a test set of 47,384 documents—as

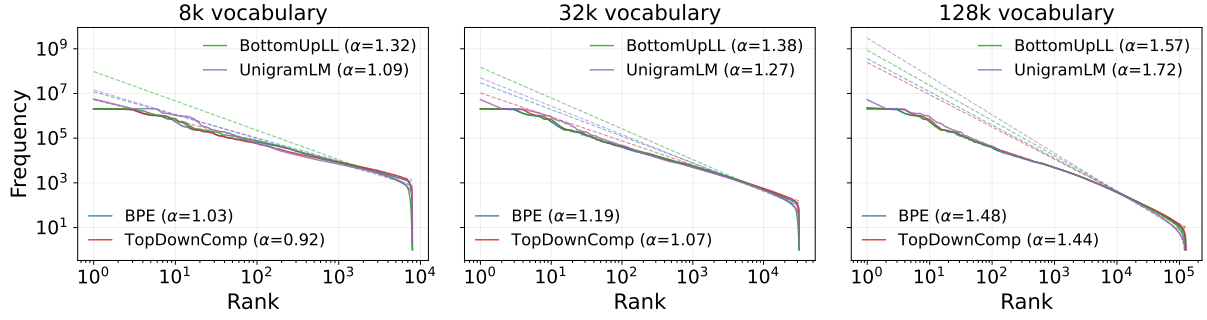


Figure 2: Frequency–rank token distributions induced by each tokeniser on the held-out test set. Solid lines are empirical curves; dashed lines are the fitted power laws, whose negated slopes are the Zipf α values in Tab. 1.

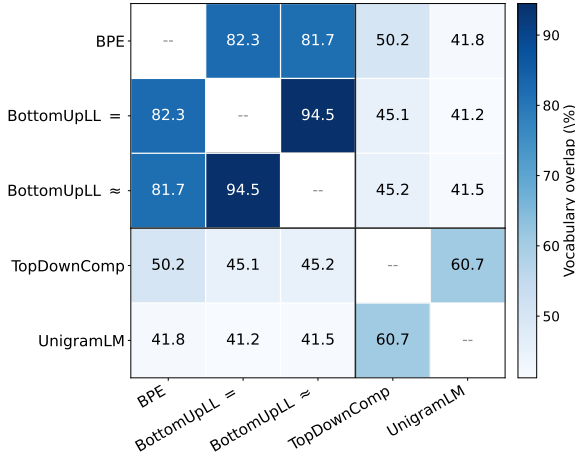


Figure 3: Pairwise vocabulary overlap at 128k vocabulary size. BottomUpLL = denotes the exact variant of this method, which scores merges with Lemma 2, while BottomUpLL $\approx$ denotes the approximate variant, which uses the PMI-based approximation of Lemma 5. See Tab. 6 in §J for a multilingual counterpart of this table.

well as the average length of the tokens in their vocabularies. Vocabulary utilisation is close to 100% at 8k and 32k, but at 128k, likelihood-based tokenisers have more unused tokens; §K traces this to the intermediate tokens that BottomUpLL creates during merging and then never uses again. Token length behaves differently across vocabulary sizes: at small vocabulary sizes, top-down methods present longer tokens, as they can retain these tokens from the initial seed vocabulary; at larger vocabulary sizes, though, bottom-up methods present longer tokens instead, as they can build these tokens from scratch through successive merges. Interestingly, across both search procedures, the likelihood-based tokenisers have longer tokens than their compression-based counterparts.

Vocabulary Composition. We now move on to analysing the composition of our tokenisers’ vocabularies. To this end, we compute their similar-

ity, defined as $|S_A \cap S_B|/|S_A|$. Fig. 3 presents these results. From this table, we see that learned vocabularies cluster primarily by search procedure: e.g., bottom-up methods share substantially more tokens with each other than with top-down methods. This indicates that a tokeniser’s search procedure has a large effect on which tokens are included in its final vocabulary. Fig. 4 in §L shows the full set-overlap structure of all five vocabularies. Furthermore, per-example qualitative segmentations for the four tokenisers are shown in §M.

7.2 Extrinsic Evaluation

BPB. Tabs. 2 and 3 present extrinsic results for both the English and multilingual settings. From these tables, we can see that bottom-up tokenisers consistently achieve lower BPB scores than top-down. This holds true for all our experimental conditions, with the only exception of the English 300M models at 32k vocabulary, where TopDownComp outperforms BPE by a negligible margin (0.0003 BPB); the likelihood-based pair (BottomUpLL vs. UnigramLM) favours bottom-up in every setting.¹⁴ Comparing tokenisers within a single search procedure, we see that the choice of objective is still relevant, albeit less strongly. Compression-based methods tend to outperform log-likelihood; but there are several exceptions where BottomUpLL outperforms all other methods.

Minimal-pair accuracy. Tabs. 2 and 3 also present scores on a minimal-pair grammaticality task, for both the English and multilingual settings. Notably, these results do not show the same clear preference as BPB for bottom-up over top-down. In fact, in the English setting, top-down methods often outperform the bottom-up ones,

¹⁴Because several of these differences are small, we additionally verify their significance with a paired document-level bootstrap over the evaluation set, which confirms the statistical significance of each of these comparisons (see details in §N).

Vocab	Tokeniser	Objective	Search	100M		300M		500M		1B	
				BLiMP $\uparrow$	BPB $\downarrow$	BLiMP $\uparrow$	BPB $\downarrow$	BLiMP $\uparrow$	BPB $\downarrow$	BLiMP $\uparrow$	BPB $\downarrow$
8k	BPE	$\mathcal{G}_{\text{comp}}$	bottom-up	.686 $\pm$ 6e-3	1.0736 $\pm$ 2e-4	.7707	.8739	.7744	.8435	–	–
	TopDownComp	$\mathcal{G}_{\text{comp}}$	top-down	.687 $\pm$ 4e-3	1.0750 $\pm$ 8e-4	.7699	.8758	.7938	.8441	–	–
	BottomUpLL	$\mathcal{G}_{\text{ll}}$	bottom-up	.703 $\pm$ 8e-3	1.0715 $\pm$ 6e-4	.7841	.8727	.7951	.8420	–	–
	UnigramLM	$\mathcal{G}_{\text{ll}}$	top-down	.698 $\pm$ 5e-3	1.0819 $\pm$ 4e-4	.7769	.8758	.7676	.8472	–	–
32k	BPE	$\mathcal{G}_{\text{comp}}$	bottom-up	.714 $\pm$ 8e-3	1.0392 $\pm$ 5e-4	.7904	.8584	.7958	.8290	–	–
	TopDownComp	$\mathcal{G}_{\text{comp}}$	top-down	.727 $\pm$ 2e-2	1.0394 $\pm$ 3e-4	.7895	.8581	.7986	.8307	–	–
	BottomUpLL	$\mathcal{G}_{\text{ll}}$	bottom-up	.717 $\pm$ 7e-3	1.0374 $\pm$ 1e-4	.7827	.8564	.7899	.8281	–	–
	UnigramLM	$\mathcal{G}_{\text{ll}}$	top-down	.719 $\pm$ 8e-3	1.0441 $\pm$ 9e-4	.7776	.8610	.8015	.8337	–	–
128k	BPE	$\mathcal{G}_{\text{comp}}$	bottom-up	.740 $\pm$ 4e-3	1.0138 $\pm$ 4e-4	.797	.8462	.813	.8210	.805	.7790 $\pm$ 4e-4
	TopDownComp	$\mathcal{G}_{\text{comp}}$	top-down	.740 $\pm$ 8e-3	1.0172 $\pm$ 4e-4	.797	.8492	.815	.8226	.816	.7816 $\pm$ 3e-4
	BottomUpLL	$\mathcal{G}_{\text{ll}}$	bottom-up	.725 $\pm$ 6e-3	1.0149 $\pm$ 9e-4	.792	.8467	.798	.8207	.799	.7803 $\pm$ 26e-4
	UnigramLM	$\mathcal{G}_{\text{ll}}$	top-down	.738 $\pm$ 7e-3	1.0298 $\pm$ 8e-4	.799	.8547	.803	.8283	.818	.7872 $\pm$ 1e-4

Table 2: Extrinsic tokenisation results for language models trained in English. Cells marked “–” were not run; 1B models were trained only at 128k vocabulary. 100M results are averaged over three random seeds; for the 1B models, BPB is likewise averaged over three seeds (42, 43, 44), while BLiMP uses a single seed. Values following $\pm$ are standard deviations across seeds. Lower is better for BPB; higher is better for BLiMP.

Tokeniser	Objective	Search	BLiMP $\uparrow$	Minimal-pair accuracy $\uparrow$					BPB $\downarrow$				
				eng	deu	spa	tur	cmn	eng	deu	spa	tur	cmn
BPE	$\mathcal{G}_{\text{comp}}$	bottom-up	.812	.974	.955	.958	.890	.799	.8080	.9426	.9025	.8936	1.2464
TopDownComp	$\mathcal{G}_{\text{comp}}$	top-down	.808	.969	.966	.956	.853	.718	.8276	.9711	.9284	.9260	1.3430
BottomUpLL	$\mathcal{G}_{\text{ll}}$	bottom-up	.808	.970	.952	.950	.873	.803	.8082	.9416	.9030	.8929	1.2436
UnigramLM	$\mathcal{G}_{\text{ll}}$	top-down	.803	.973	.964	.959	.897	.728	.8311	.9683	.9303	.9252	1.3575

Table 3: Extrinsic tokenisation results for 1B parameter models trained in a multilingual setting. Minimal-pair accuracy is evaluated on BLiMP (Warstadt et al., 2020), MultiBLiMP (eng, deu, spa, tur; Jumelet et al., 2026), and ZhoBLiMP (cmn; Liu et al., 2026). Lower is better for BPB; higher is better for minimal-pair accuracy.

with UnigramLM outperforming BottomUpLL and TopDownComp outperforming BPE for the 1B models. In the multilingual setting, no consistent pattern arises either, and all tokenisers perform best in at least one language. Our results in German, Spanish, and Turkish, however, suggest that top-down methods may be helpful for morphologically rich languages. More broadly, our multilingual minimal-pair accuracy results suggest different tokenisers induce different inductive biases, which will in turn be uniquely suited to different languages; we leave this direction to future work.

8 Conclusion

Our paper disentangles two important design choices in tokeniser learning algorithms: the objective used to score a tokeniser and the search procedure used to optimise it. We analyse two objectives (compression vs. log-likelihood) and search procedures (bottom-up vs. top-down), introducing two new tokenisers in the process: BottomUpLL and TopDownComp. In nearly all experimental conditions, language models trained with bottom-up tokenisers outperformed top-down ones in terms of

bits-per-byte. Our intrinsic evaluations also highlighted a surprising pattern: bottom-up tokenisers produced both more compressed and higher log-likelihood token-strings than top-down tokenisers, independent of the used objective function. The used objective still affects the resulting tokeniser, though, with log-likelihood objective leaving a clear impact on the empirical rank–frequency distribution of its tokens. Overall, our results show that a tokeniser’s effect on language modelling cannot be explained by the objective alone, and that the used search procedure is an important component of tokeniser design. We hope future work will explore the impact of other optimisation objectives and search procedures¹⁵ in language modelling.

Limitations

We list a few limitations with our study here.

Scale. First, we only train language models up to 1B parameters, and results could shift for larger models or longer training horizons.

¹⁵Investigating the search procedures of, e.g., Tempus et al. (2026) or Chizhov et al. (2024).

Languages. Second, our multilingual experiments cover five selected languages: English, German, Spanish, Turkish, and Chinese. Four of these use the Latin script and all are relatively high-resource, so our multilingual results should be read as applying to these five languages rather than to multilingual tokenisation in general. How our findings transfer to low-resource languages, and to scripts with markedly different orthographic conventions, remains an open question that we leave to future work.

Single seed at scale. Third, for computational reasons, the 300M and 500M models are trained with a single seed per configuration, as are the multilingual 1B models; the 100M and English 1B models are averaged over three seeds; small differences in results across tokenisers should thus be interpreted with caution.

Tokeniser-training corpus. Fourth, in all our experiments, each tokeniser is trained on the same corpus as the language model trained on top of it. We thus do not analyse the impact of training models on data that differ in domain or style from its tokeniser.

Approximate top-down deletion scores. Fifth, as discussed in §4.2, top-down deletion costs rely on the *local replacement approximation*, whereas bottom-up merge gains are exact for the current step. We assess the impact of this approximation by, at a small scale, training tokenisers for which we compute exact deletion scoring. We find that the local-replacement-approximation tokenisers present an 81.5% vocabulary overlap with these exact ones. Whether this holds at the vocabulary sizes used in our main experiments, however, remains open. A second approximation we rely on for top-down methods is that we *prune multiple tokens* per round rather than one at a time. We assess the impact of this approximation by re-training the top-down tokenisers with pruning rates of 1% and 0.1% of the vocabulary per round, instead of the 10% used in our main experiments: the resulting vocabularies overlap with the 10% ones by over 97% for UnigramLM and over 99% for TopDownComp, suggesting that this approximation has a limited effect on the learned vocabulary. Details for both experiments are in §O.

Evaluation. Finally, we evaluate models with BPB and minimal-pair grammatical benchmarks (BLiMP, MultiBLiMP, ZhoBLiMP). We do not

measure other downstream task performance, reasoning ability, or long-context behaviour; how our models perform in those settings is thus left open.

Acknowledgements

We thank Philip Whittington for helpful discussions and feedback throughout this project, and the anonymous reviewers for their constructive comments. This work was supported as part of the “Swiss AI initiative” by a grant from the Swiss National Supercomputing Centre (CSCS) under project ID a0229 on Alps. Tiago and Ahmetcan also thank the Data Analytics Lab at ETH Zürich for supporting this research and for financing conference participation, as well as for conversations with other lab members during the work.

References

- Pavel Chizhov, Catherine Arnett, Elizaveta Korotkova, and Ivan P. Yamshchikov. 2024. [BPE gets picky: Efficient vocabulary refinement during tokenizer training](#). In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 16587–16604, Miami, Florida, USA. Association for Computational Linguistics.
- Philip Gage. 1994. [A new algorithm for data compression](#). *C Users Journal*, 12(2):23–38.
- Hugging Face. 2026. WordPiece tokenization. Chapter 6.6 of the Hugging Face LLM Course, <https://huggingface.co/learn/llm-course/en/chapter6/6>. Accessed August 2026.
- Jaap Jumelet, Leonie Weissweiler, Joakim Nivre, and Arianna Bisazza. 2026. [MultiBLiMP 1.0: A massively multilingual benchmark of linguistic minimal pairs](#). *Transactions of the Association for Computational Linguistics*, 14:193–216.
- Violeta Kastreva, Philip Whittington, Dennis Komm, and Tiago Pimentel. 2026. [Tokenisation over bounded alphabets is hard](#). In *The Fourteenth International Conference on Learning Representations*.
- László Kozma and Johannes Voderholzer. 2024. [Theoretical analysis of byte-pair encoding](#). *arXiv preprint arXiv:2411.08671*.
- Taku Kudo. 2018. [Subword regularization: Improving neural network translation models with multiple subword candidates](#). In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 66–75.
- Pietro Lesci, Clara Meister, Thomas Hofmann, Andreas Vlachos, and Tiago Pimentel. 2025. [Causal estimation of tokenisation bias](#). In *Proceedings of the 63rd Annual Meeting of the Association for Computational*

- Linguistics (Volume 1: Long Papers)*, pages 28325–28340, Vienna, Austria. Association for Computational Linguistics.
- Yikang Liu, Yeting Shen, Hongao Zhu, Lilong Xu, Zhiheng Qian, Siyuan Song, Kejia Zhang, Jialong Tang, Pei Zhang, Baosong Yang, Rui Wang, and Hai Hu. 2026. [A systematic assessment of language models with linguistic minimal pairs in Chinese](#). *Transactions of the Association for Computational Linguistics*, 14:755–771.
- Anton Lozhkov, Loubna Ben Allal, Leandro von Werra, and Thomas Wolf. 2024. [FineWeb-Edu: the finest collection of educational content](#).
- Clara Meister. 2026. [UnigramLM: An attempt at writing the missing manual](#). In *Proceedings of the Fourteenth International Conference on Learning Representations*. ICLR 2026 Blogpost Track.
- OpenAI, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, Che Chang, and 107 others. 2025. [gpt-oss-120b & gpt-oss-20b model card](#). *arXiv preprint arXiv:2508.10925*.
- Guilherme Penedo, Hynek Kydlíček, Vinko Sabolčec, Bettina Messmer, Negar Foroutan, Amir Hossein Kargaran, Colin Raffel, Martin Jaggi, Leandro Von Werra, and Thomas Wolf. 2025. [FineWeb2: One pipeline to scale them all – adapting pre-training data processing to every language](#). *arXiv preprint arXiv:2506.20920*.
- Craig W Schmidt, Varshini Reddy, Haoran Zhang, Alec Alameddine, Omri Uzan, Yuval Pinter, and Chris Tanner. 2024. [Tokenization is more than compression](#). In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 678–702.
- Mike Schuster and Kaisuke Nakajima. 2012. [Japanese and Korean voice search](#). In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5149–5152.
- Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. [Neural machine translation of rare words with subword units](#). In *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 1: long papers)*, pages 1715–1725.
- Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, S. H. Cai, Yuan Cao, Ziwei Chai, Y. Charles, H. S. Che, Cheng Chen, Guanduo Chen, Huarong Chen, Jia Chen, Jianlong Chen, Jun Chen, Kefan Chen, Liang Chen, Ruijue Chen, Xinhao Chen, and 318 others. 2026. [Kimi K2.5: Visual agentic intelligence](#). *arXiv preprint arXiv:2602.02276*.
- Qwen Team. 2026. [Qwen3.5-Omni technical report](#). *arXiv preprint arXiv:2604.15804*.
- Jan Tempus, Philip Whittington, Craig W. Schmidt, Dennis Komm, and Tiago Pimentel. 2026. [Tokenisation via convex relaxations](#). *arXiv preprint arXiv:2605.22821*.
- David Vilar and Marcello Federico. 2021. [A statistical extension of byte-pair encoding](#). In *Proceedings of the 18th International Conference on Spoken Language Translation (IWSLT 2021)*, pages 263–275.
- Alex Warstadt, Alicia Parrish, Haokun Liu, Anhad Mohananey, Wei Peng, Sheng-Fu Wang, and Samuel R. Bowman. 2020. [BLiMP: The benchmark of linguistic minimal pairs for English](#). *Transactions of the Association for Computational Linguistics*, 8:377–392.
- Philip Whittington, Gregor Bachmann, and Tiago Pimentel. 2025. [Tokenisation is NP-complete](#). In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 28133–28153.
- Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, and 3 others. 2020. [Transformers: State-of-the-art natural language processing](#). In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.
- Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, and 12 others. 2016. [Google’s neural machine translation system: Bridging the gap between human and machine translation](#). *arXiv preprint arXiv:1609.08144*.

A Proof of Lemma 1

Lemma 1. *The change in compression due to merging a token-pair can be computed as:*

$$\Delta_{s^1, s^2}^{\mathfrak{G}_{\text{comp}}(\cdot, \mathcal{D})} = n_{s^1, s^2}^{\mathcal{D}} \quad (9)$$

where $n_{s^1, s^2}^{\mathcal{D}}$ denotes the number of non-overlapping $\langle s^1, s^2 \rangle$ sequences in $\mathcal{D}$.

Proof. To prove this is the case, first note that replacing a token-pair $m = \langle s^1, s^2 \rangle$ with merged token $s^1 \circ s^2$ saves exactly 1 token. As the number of non-overlapping occurrences of a token-pair is also the number of possible merges, it is equivalent to the compression this token-pair would achieve:

$$\begin{aligned} \mathfrak{G}(\text{tok}_{\uparrow}[\mathbf{m}_{<k}], \mathcal{D}) - \mathfrak{G}(\text{tok}_{\uparrow}[\mathbf{m}_{<k} \circ \langle s^1, s^2 \rangle], \mathcal{D}) \\ = n_{s^1, s^2}^{\mathcal{D}}. \end{aligned} \quad (16)$$

We can now trivially conclude the proof:

$$m^* = \underset{m \in \Sigma^+ \times \Sigma^+}{\text{argmin}} \mathfrak{G}(\text{tok}_{\uparrow}[\mathbf{m}_{<k} \circ m], \mathcal{D}) \quad (17a)$$

$$= \underset{m \in \Sigma^+ \times \Sigma^+}{\text{argmax}} -\mathfrak{G}(\text{tok}_{\uparrow}[\mathbf{m}_{<k} \circ m], \mathcal{D}) \quad (17b)$$

$$= \underset{m \in \Sigma^+ \times \Sigma^+}{\text{argmax}} \mathfrak{G}(\text{tok}_{\uparrow}[\mathbf{m}_{<k}], \mathcal{D}) \quad (17c)$$

$$\begin{aligned} & - \mathfrak{G}(\text{tok}_{\uparrow}[\mathbf{m}_{<k} \circ m], \mathcal{D}) \\ & = \underset{\langle s^1, s^2 \rangle \in \Sigma^+ \times \Sigma^+}{\text{argmax}} n_{s^1, s^2}^{\mathcal{D}}. \end{aligned} \quad (17d)$$

This concludes the proof. $\square$

B Updating Counts in bottom-up Procedures

Both BPE and BottomUpLL repeatedly select a token-pair based on its merge gain on the current tokenised corpus. A naive implementation would recompute token and token-pair counts after each merge, which is expensive on large corpora. Instead, our implementation maintains these counts incrementally and updates only the parts of the corpus affected by the selected merge.

The corpus is represented as a collection of unique word types, each with an associated frequency. Each word type stores its current tokenisation using a linked token structure: every token stores pointers to its predecessor and successor, allowing merges to be applied locally without reconstructing the full token string. We maintain a mapping

$$\begin{aligned} \text{pair_to_words} : \langle s^1, s^2 \rangle \\ \mapsto \{\text{word types containing } \langle s^1, s^2 \rangle\}, \end{aligned} \quad (18)$$

which maps each token-pair to the word types in which that pair currently occurs. We also maintain a reverse index from tokens to the token-pairs containing them, which allows us to identify which candidate scores may be affected after a merge.

When a merge $m = \langle s^1, s^2 \rangle$ is selected, write

$$s^{\text{new}} \stackrel{\text{def}}{=} s^1 \circ s^2.$$

Each affected word type is scanned through its linked token sequence, and non-overlapping occurrences of $\langle s^1, s^2 \rangle$ are replaced by s^{new} . The non-overlap condition is enforced during this scan: once an occurrence has been merged, the scan skips over the newly created token, so overlapping occurrences of the same pair are not counted twice.

Locally, an occurrence in context

$$\langle s^{\ell}, s^1, s^2, s^r \rangle \quad (19)$$

is replaced by

$$\langle s^{\ell}, s^{\text{new}}, s^r \rangle, \quad (20)$$

where s^{ℓ} and s^r denote the immediate left and right neighbours, if they exist. Therefore, only token-pairs in this local neighbourhood can change. The old token-pairs

$$\langle s^{\ell}, s^1 \rangle, \quad \langle s^1, s^2 \rangle, \quad \langle s^2, s^r \rangle \quad (21)$$

are removed, and the new token-pairs

$$\langle s^{\ell}, s^{\text{new}} \rangle, \quad \langle s^{\text{new}}, s^r \rangle \quad (22)$$

are added. Boundary cases are handled by omitting updates involving missing neighbours.

For one merged occurrence, the affected pair counts are updated as

$$n_{s^{\ell}, s^1}^{\mathcal{D}} \leftarrow n_{s^{\ell}, s^1}^{\mathcal{D}} - 1, \quad (23a)$$

$$n_{s^1, s^2}^{\mathcal{D}} \leftarrow n_{s^1, s^2}^{\mathcal{D}} - 1, \quad (23b)$$

$$n_{s^2, s^r}^{\mathcal{D}} \leftarrow n_{s^2, s^r}^{\mathcal{D}} - 1, \quad (23c)$$

$$n_{s^{\ell}, s^{\text{new}}}^{\mathcal{D}} \leftarrow n_{s^{\ell}, s^{\text{new}}}^{\mathcal{D}} + 1, \quad (23d)$$

$$n_{s^{\text{new}}, s^r}^{\mathcal{D}} \leftarrow n_{s^{\text{new}}, s^r}^{\mathcal{D}} + 1. \quad (23e)$$

When word types have frequencies, these increments and decrements are weighted by the corresponding word-type frequency.

For BottomUpLL, we also update unigram counts, since the merge score in Lemma 2 depends on both token counts and pair counts. If $n_{s^1, s^2}^{\mathcal{D}}$

denotes the frequency-weighted number of non-overlapping occurrences of the selected pair that are actually merged, then the unigram counts are updated as

$$n_{s^1}^D \leftarrow n_{s^1}^D - n_{s^1, s^2}^D, \quad (24a)$$

$$n_{s^2}^D \leftarrow n_{s^2}^D - n_{s^1, s^2}^D, \quad (24b)$$

$$n_{s^{\text{new}}}^D \leftarrow n_{s^1, s^2}^D, \quad (24c)$$

$$N^D \leftarrow N^D - n_{s^1, s^2}^D. \quad (24d)$$

For BPE, the affected candidate scores are simply the updated pair counts, as in Lemma 1. For BottomUpLL, affected candidate scores are then recomputed using the local likelihood-change expression in Lemma 2.

The priority queue of candidate token-pairs is maintained lazily. After a merge, the selected pair is removed from the pair-to-word index, affected reverse-index entries are updated, and affected token-pairs are pushed back onto the heap with fresh scores. Old heap entries are not removed immediately. Instead, stale entries are rejected when popped: the implementation checks the current count and score, and discards the heap entry if it no longer matches the current state. The heap is also periodically rebuilt to control the accumulation of stale entries.

This gives an incremental implementation in which the cost of a merge is proportional to the number of affected word types and local token updates, plus the cost of heap maintenance. In practice, this is much cheaper than recomputing all token and token-pair counts over the full corpus after every merge.

C Proof of Lemma 2

Lemma 2. *The change in log-likelihood due to merging a token-pair can be computed as:*

$$\Delta_{s^1, s^2}^{\mathcal{O}_{\Pi}(\cdot, D)} = \quad (10)$$

$$\begin{aligned} & (n_{s^1}^D - n_{s^1, s^2}^D) \log(n_{s^1}^D - n_{s^1, s^2}^D) - n_{s^1}^D \log n_{s^1}^D \\ & + (n_{s^2}^D - n_{s^1, s^2}^D) \log(n_{s^2}^D - n_{s^1, s^2}^D) - n_{s^2}^D \log n_{s^2}^D \\ & + n_{s^1, s^2}^D \log n_{s^1, s^2}^D \\ & - (N^D - n_{s^1, s^2}^D) \log(N^D - n_{s^1, s^2}^D) + N^D \log N^D \end{aligned}$$

Proof. We prove the result by explicitly writing the corpus log-likelihood before and after applying the merge. Let the current tokenised corpus contain N^D tokens in total. For each token $\tilde{s}$, let $n_{\tilde{s}}^D$ denote

its corpus count. Under the unigram token model, the empirical probability of $\tilde{s}$ is

$$p_{\theta}(\tilde{s}) = \frac{n_{\tilde{s}}^D}{N^D}. \quad (25)$$

Therefore, the corpus log-likelihood before applying the merge is

$$\mathcal{L}_{\text{before}} = \sum_{\tilde{s}} n_{\tilde{s}}^D \log p_{\theta}(\tilde{s}) \quad (26a)$$

$$= \sum_{\tilde{s}} n_{\tilde{s}}^D \log \frac{n_{\tilde{s}}^D}{N^D} \quad (26b)$$

$$= \sum_{\tilde{s}} n_{\tilde{s}}^D \log n_{\tilde{s}}^D - N^D \log N^D. \quad (26c)$$

Now consider merging the token-pair $\langle s^1, s^2 \rangle$ into the new token $s^1 \circ s^2$. For readability, write

$$\kappa \stackrel{\text{def}}{=} n_{s^1, s^2}^D, \quad (27)$$

Assuming $s^1 \neq s^2$, applying this merge changes only the counts of s^1 , s^2 , and the new token $s^1 \circ s^2$. Specifically,

$$n_{s^1}^D \mapsto n_{s^1}^D - \kappa, \quad (28a)$$

$$n_{s^2}^D \mapsto n_{s^2}^D - \kappa, \quad (28b)$$

$$n_{s^1 \circ s^2}^D \mapsto \kappa. \quad (28c)$$

All other token counts remain unchanged. Since each merged occurrence replaces two tokens by one token, the total token count changes from N^D to $N^D - \kappa$. The log-likelihood after the merge is therefore

$$\begin{aligned} \mathcal{L}_{\text{after}} = & \sum_{\tilde{s} \notin \{s^1, s^2, s^1 \circ s^2\}} n_{\tilde{s}}^D \log n_{\tilde{s}}^D \quad (29) \\ & + (n_{s^1}^D - \kappa) \log(n_{s^1}^D - \kappa) \\ & + (n_{s^2}^D - \kappa) \log(n_{s^2}^D - \kappa) \\ & + \kappa \log \kappa - (N^D - \kappa) \log(N^D - \kappa). \end{aligned}$$

and the log-likelihood change due to the merge is

$$\Delta_{s^1, s^2}^{\mathcal{O}_{\Pi}(\cdot, D)} = \mathcal{L}_{\text{after}} - \mathcal{L}_{\text{before}}. \quad (30)$$

Substituting Eqs. (26c) and (30), all unchanged token-count terms cancel, leaving

$$\Delta_{s^1, s^2}^{\mathcal{O}_{\Pi}(\cdot, D)} = \quad (31)$$

$$\begin{aligned} & (n_{s^1}^D - \kappa) \log(n_{s^1}^D - \kappa) \\ & - n_{s^1}^D \log n_{s^1}^D \\ & + (n_{s^2}^D - \kappa) \log(n_{s^2}^D - \kappa) \\ & - n_{s^2}^D \log n_{s^2}^D \\ & + \kappa \log \kappa \\ & - (N^D - \kappa) \log(N^D - \kappa) + N^D \log N^D. \end{aligned}$$

which completes the proof. $\square$

D Proof of Lemma 3

Lemma 3. *Under a local replacement approximation, the estimated increase in the negative log-likelihood objective due to deleting a token $s \in \mathcal{S} \setminus \Sigma$ can be approximated as:*

$$\Delta_s^{\mathfrak{G}_{\text{ll}}(\cdot, \mathcal{D})} \approx n_s^{\mathcal{D}} (\log p_{\theta}(s) - \log p_{\theta}(s_s^{\text{rep}})), \quad (11)$$

where p_{θ} is the unigram model at step $k - 1$, and s_s^{rep} is s 's log-probability-optimal replacement segmentation, i.e., $s_s^{\text{rep}} \stackrel{\text{def}}{=} \text{tok}_{\rightarrow}[\mathcal{S}_{k-1} \setminus \{s\}](s)$.

Proof. We prove the result under the local replacement approximation stated in the lemma. That is, when scoring the deletion of s , we keep the current token probabilities p_{θ} fixed, keep all other segmentation decisions fixed, and approximate the effect of deletion by replacing each expected occurrence of s with a replacement segmentation under $\mathcal{S} \setminus \{s\}$.

For UnigramLM, token usage is measured by expected counts under the current posterior over valid segmentations. Thus, $n_s^{\mathcal{D}}$ is the expected number of times s is used in the corpus. Before deleting s , each such expected occurrence contributes

$$-\log p_{\theta}(s) \quad (32)$$

to the negative log-likelihood objective.

After deleting s , the token can no longer be used as a single token. Under the local replacement approximation, each occurrence of s is replaced by the highest-probability valid segmentation of the same string under the reduced vocabulary:

$$s_s^{\text{rep}} \stackrel{\text{def}}{=} \underset{\substack{\tilde{s} \in (\mathcal{S} \setminus \{s\})^* \\ s \stackrel{\circ}{=} \tilde{s}}}{\text{argmax}} p_{\theta}(\tilde{s}). \quad (33)$$

Since p_{θ} is a unigram model, the probability of this replacement token-string is

$$p_{\theta}(s_s^{\text{rep}}) = \prod_{\tilde{s} \in s_s^{\text{rep}}} p_{\theta}(\tilde{s}). \quad (34)$$

After replacement, each expected occurrence therefore contributes

$$-\log p_{\theta}(s_s^{\text{rep}}) \quad (35)$$

to the negative log-likelihood objective.

Hence, the estimated increase in negative log-likelihood for one expected occurrence of s is

$$\begin{aligned} & -\log p_{\theta}(s_s^{\text{rep}}) - (-\log p_{\theta}(s)) \\ & = \log p_{\theta}(s) - \log p_{\theta}(s_s^{\text{rep}}). \end{aligned} \quad (36)$$

Multiplying this per-occurrence increase by the expected number of occurrences $n_s^{\mathcal{D}}$ gives

$$\Delta_s^{\mathfrak{G}_{\text{ll}}(\cdot, \mathcal{D})} \approx n_s^{\mathcal{D}} (\log p_{\theta}(s) - \log p_{\theta}(s_s^{\text{rep}})). \quad (37)$$

This proves the stated local approximation. $\square$

E Proof of Lemma 4

Lemma 4. *Under a local replacement approximation, the estimated change in compression due to deleting a token $s \in \mathcal{S} \setminus \Sigma$ can be approximated as:*

$$\Delta_s^{\mathfrak{G}_{\text{comp}}(\cdot, \mathcal{D})} = n_s^{\mathcal{D}} (|s_s^{\text{rep}}| - 1), \quad (12)$$

where s_s^{rep} is the compression-optimal replacement of s after deletion: $s_s^{\text{rep}} \stackrel{\text{def}}{=} \text{tok}_{\rightarrow}[\mathcal{S}_{k-1} \setminus \{s\}](s)$.

Proof. Let $\mathcal{S}$ be the current vocabulary, and suppose the corpus has already been segmented using the TopDownComp encoding rule:

$$\text{tok}_{\rightarrow}[\mathcal{S}](c) = \underset{\substack{s \in \mathcal{S}^* \\ c \stackrel{\circ}{=} s}}{\text{argmin}} |s|. \quad (38)$$

Now consider deleting a token $s \in \mathcal{S} \setminus \Sigma$. Before deletion, each current occurrence of s contributes exactly one token to the corpus token count. Therefore, the total contribution of all current occurrences of s before deletion is

$$C_{\text{before}}(s) = n_s^{\mathcal{D}}. \quad (39)$$

After deleting s , the token s can no longer appear as a single token. Under the local replacement calculation, each current occurrence of s is independently replaced by its shortest decomposition under the reduced vocabulary $\mathcal{S} \setminus \{s\}$, denoted s_s^{rep} :

$$s_s^{\text{rep}} = \underset{\substack{s' \in (\mathcal{S} \setminus \{s\})^* \\ s \stackrel{\circ}{=} s'}}{\text{argmin}} |s'|. \quad (40)$$

Thus, under this local replacement calculation, each previous occurrence of s contributes $|s_s^{\text{rep}}|$ tokens instead of one token. The total contribution of these occurrences after deletion is

$$C_{\text{after}}(s) = n_s^{\mathcal{D}} |s_s^{\text{rep}}|. \quad (41)$$

Therefore, the estimated increase in corpus length caused by locally replacing all current occurrences of s is

$$\Delta_s^{\mathfrak{G}_{\text{comp}}(\cdot, \mathcal{D})} = C_{\text{after}}(s) - C_{\text{before}}(s) \quad (42a)$$

$$= n_s^{\mathcal{D}} |s_s^{\text{rep}}| - n_s^{\mathcal{D}} \quad (42b)$$

$$= n_s^{\mathcal{D}} (|s_s^{\text{rep}}| - 1). \quad (42c)$$

which completes this proof. $\square$

F Proof of Lemma 5

Lemma 5. *Under a first-order Taylor approximation, the change in log-likelihood due to merging a token-pair can be approximated as:*

$$\Delta_{s^1, s^2}^{\mathcal{G}_{\Pi}(\cdot, \mathcal{D})} \approx n_{s^1, s^2}^{\mathcal{D}} (\text{PMI}_{\text{adj}}(s^1, s^2) - 1). \quad (14)$$

Proof. As in §C, we write $\kappa \stackrel{\text{def}}{=} n_{s^1, s^2}^{\mathcal{D}}$. Starting from Eq. (31), we have

$$\begin{aligned} \Delta_{s^1, s^2}^{\mathcal{G}_{\Pi}(\cdot, \mathcal{D})} = & \quad (43) \\ & (n_{s^1}^{\mathcal{D}} - \kappa) \log(n_{s^1}^{\mathcal{D}} - \kappa) - n_{s^1}^{\mathcal{D}} \log n_{s^1}^{\mathcal{D}} \\ & + (n_{s^2}^{\mathcal{D}} - \kappa) \log(n_{s^2}^{\mathcal{D}} - \kappa) - n_{s^2}^{\mathcal{D}} \log n_{s^2}^{\mathcal{D}} \\ & + \kappa \log \kappa \\ & - (N^{\mathcal{D}} - \kappa) \log(N^{\mathcal{D}} - \kappa) + N^{\mathcal{D}} \log N^{\mathcal{D}}. \end{aligned}$$

Now, define $f(a) = a \log a$, which implies $f'(a) = \log a + 1$. A first-order Taylor expansion of $f(a)$ around point x gives

$$f(a) \approx f(x) + (a - x)f'(x) \quad (44a)$$

$$f(x - \kappa) \approx f(x) + (x - \kappa - x)f'(x) \quad (44b)$$

$$= f(x) - \kappa f'(x) \quad (44c)$$

$$= x \log x - \kappa(\log x + 1) \quad (44d)$$

This approximation is accurate whenever $\kappa \ll x$: the neglected remainder is $\frac{1}{2}\kappa^2 f''(\xi) = O(\kappa^2/x)$, which is small relative to the retained term $\kappa(\log x + 1)$ exactly in that regime. Now, choosing $x = n_{s^1}^{\mathcal{D}}$, we get

$$\begin{aligned} f(n_{s^1}^{\mathcal{D}} - \kappa) & \\ &= (n_{s^1}^{\mathcal{D}} - \kappa) \log(n_{s^1}^{\mathcal{D}} - \kappa) \quad (45a) \\ &\approx n_{s^1}^{\mathcal{D}} \log n_{s^1}^{\mathcal{D}} - \kappa (\log n_{s^1}^{\mathcal{D}} + 1). \quad (45b) \end{aligned}$$

which implies

$$\begin{aligned} (n_{s^1}^{\mathcal{D}} - \kappa) \log(n_{s^1}^{\mathcal{D}} - \kappa) - n_{s^1}^{\mathcal{D}} \log n_{s^1}^{\mathcal{D}} & \quad (46) \\ \approx -\kappa (\log n_{s^1}^{\mathcal{D}} + 1). \end{aligned}$$

With similar results for $x = n_{s^2}^{\mathcal{D}}$ and $x = N^{\mathcal{D}}$.

Substituting these terms into Eq. (43), we get

$$\begin{aligned} \Delta_{s^1, s^2}^{\mathcal{G}_{\Pi}(\cdot, \mathcal{D})} \approx & -\kappa (\log n_{s^1}^{\mathcal{D}} + 1) - \kappa (\log n_{s^2}^{\mathcal{D}} + 1) \\ & + \kappa \log \kappa + \kappa (\log N^{\mathcal{D}} + 1) \quad (47a) \end{aligned}$$

$$= \kappa \left(\log \frac{N^{\mathcal{D}} \kappa}{n_{s^1}^{\mathcal{D}} n_{s^2}^{\mathcal{D}}} - 1 \right) \quad (47b)$$

$$= \kappa (\text{PMI}_{\text{adj}}(s^1, s^2) - 1). \quad (47c)$$

which completes the proof. $\square$

G Tokeniser Training Details

All four tokenisers use the same preprocessing pipeline, implemented with the HuggingFace tokenizers library. Text is normalised with NFC, and pretokenised with a byte-level pretokeniser, `ByteLevel(add_prefix_space=False, trim_offsets=True, use_regex=True)`, which applies the GPT-2 pretokenisation regex. The corresponding decoder and post-processor are:

```
ByteLevel(add_prefix_space=True,
          trim_offsets=True, use_regex=True)
ByteLevel(add_prefix_space=True,
          trim_offsets=False, use_regex=True)
```

respectively. This choice is deliberately standard. Because pretokenisation strongly constrains which token boundaries a tokeniser can learn, holding it fixed is necessary for the objective and the search procedure to be the only varying factors in our comparison. We use byte-level fallback throughout, so every string remains encodable and no `<unk>` tokens are required.

H Hyperparameters and Architecture

All language models follow a Llama-style decoder-only architecture. Tab. 4 summarises the per-size architectural settings. All models use SiLU activations, grouped-query attention with separate K/V head counts, RMSNorm, and rotary positional embeddings (RoPE, $\theta = 10,000$).

Optimisation. We train models with AdamW using learning rate 3×10^{-4} , $\beta_1 = 0.9$, $\beta_2 = 0.95$, weight decay 0.1, and a maximum gradient norm of 1.0. The learning-rate schedule is warmup–stable–decay with 2,000 warmup steps, a stable phase, and 10,000 linear-decay steps, ending with a final learning rate of 10% of its peak value.

Training setup. All models are trained at sequence length 2,048 in bf16-mixed precision with FlashAttention 2. The per-device batch size is 32, and gradient accumulation is adjusted per model size to match a fixed effective batch size. Following Chinchilla-optimal scaling, each model is trained on roughly $20\times$ its parameter count in tokens. Training uses a single fixed random seed (42), except for the 100M setting and the English 1B models at 128k vocabulary, where we train three seeds (42, 43, 44) per configuration. All experiments are implemented in PyTorch with our open-source training stack.

Size	Hidden	Intermediate	Heads	KV heads	Layers	Tied
100M	576	1,536	9	3	30	✓
300M	960	2,560	15	5	32	✓
500M	1280	3,456	16	4	26	✓
1B	2048	5,632	32	4	22	✓

Table 4: Per-size architecture for the Llama-style language models. “Tied” denotes weight tying between input and output embeddings.

I Detailed Metric Definitions

This section gives precise definitions for every metric used in the paper. Let $\mathcal{D}$ denote a held-out corpus, N_{bytes} its size in UTF-8 bytes, and $N^{\mathcal{D}}$ the number of tokens produced by a given tokeniser on $\mathcal{D}$.

Extrinsic. We evaluate models based on two extrinsic metrics. **Bits per byte (BPB)**, defined as $-\log_2 P_{\theta}(\mathcal{D})/N_{\text{bytes}}$: the trained model’s negative log-likelihood on $\mathcal{D}$, normalised by the byte count. Lower is better. Because the denominator is a property of the raw text (and hence identical across tokenisers), BPB is directly comparable across tokenisers; token-level perplexity, by contrast, is biased against tokenisers that produce shorter sequences. **Minimal-pair grammatical accuracy**, which, for an acceptable-unacceptable sentence pair, evaluates the model as correct if it assigns higher per-token log-probability to the acceptable sentence. It then averages this values across a corpus of such pairs. We use BLiMP (Warstadt et al., 2020) for English, MultiBLiMP (Jumelet et al., 2026) for English, German, Spanish, and Turkish, and ZhoBLiMP (Liu et al., 2026) for Chinese. For this metric, higher is better.

Intrinsic. We evaluate models on 5 types of intrinsic metrics. **Compression** is $\mathfrak{G}_{\text{comp}} = N^{\mathcal{D}}$, i.e., the corpus token count, where lower means more compact. **Bytes per token (BPT)** is defined as $N_{\text{bytes}}/N^{\mathcal{D}}$, the average number of source bytes captured by one token, where higher means more compact. **Vocab Util** is the fraction of vocabulary IDs that appear at least once in $\mathcal{D}$. **Token length** (Tok Len) is the average character length of the entries in the learned vocabulary, with each entry counted once, unweighted by corpus frequency. (Unigram) **Entropy** is defined here as $H = -\sum_s p_{\theta}(s) \log_2 p_{\theta}(s)$, where $p_{\theta}(s)$ is a unigram language model trained on token counts. **Unigram log-likelihood objective** is

$\mathfrak{G}_{\text{ll}} = H \cdot N^{\mathcal{D}}$, where lower suggests a better fit under the empirical unigram model. **Zipf α** is the negated slope of a log-log OLS fit of frequency vs. rank, with tokens sorted in descending frequency; higher α means a steeper distribution. **Coverage 50%** is the smallest ℓ such that the ℓ most frequent tokens together account for at least half of all token occurrences in $\mathcal{D}$; smaller ℓ means usage is concentrated on a small core.

Vocabulary overlap. Finally, we also compute vocabulary overlap between tokenisers. For two tokenisers A and B , their overlap is measured as $|\mathcal{S}_A \cap \mathcal{S}_B|/|\mathcal{S}_A|$, where $\mathcal{S}_A$ and $\mathcal{S}_B$ are their learned vocabularies. This measure is symmetric for tokenisers with equal vocabulary size.

J English vs. Multilingual Intrinsic Comparison

This appendix reports the multilingual counterpart of Tab. 1 and Fig. 3, using the multilingual tokenisers and the concatenation of the five per-language test splits (220,423 documents: 47,384 English, 54,969 German, 54,489 Spanish, 41,986 Turkish, 21,595 Chinese).

Compression gaps widen in the multilingual setting. Tab. 5 shows each tokenisers compression in both English-only and multilingual corpora. In English, UnigramLM produces 18.5% more tokens than BPE and TopDownComp 7.4% more. In multilingual, the top-down gap roughly doubles: UnigramLM +39.6%, TopDownComp +26.5%. BottomUpLL remains close to BPE in both settings (+0.5% in English, +0.9% in multilingual). The bottom-up vs. top-down separation observed in the main text therefore grows under a shared multilingual vocabulary budget.

Distributional shape changes with the evaluation corpus. Tab. 5 reports the full intrinsic statistics in both settings. All four token distributions flatten in the multilingual setting: Zipf α values

Setting	Tokeniser	Objective	Search	$\mathcal{C}_{\text{comp}} \downarrow$	$\mathcal{C}_{\text{ll}} \downarrow$	Entropy	Zipf α	BPT $\uparrow$	Tok Len	Vocab Util $\uparrow$	Coverage 50%
English	BPE	$\mathcal{C}_{\text{comp}}$	bottom-up	46,859,565	5.216×10^8	11.132	1.484	4.886	7.06	98.9%	204
	TopDownComp	$\mathcal{C}_{\text{comp}}$	top-down	50,312,483	5.484×10^8	10.900	1.443	4.551	6.01	99.8%	149
	BottomUpLL	$\mathcal{C}_{\text{ll}}$	bottom-up	47,111,733	5.203×10^8	11.044	1.574	4.860	7.48	96.8%	191
	UnigramLM	$\mathcal{C}_{\text{ll}}$	top-down	55,546,423	5.552×10^8	9.996	1.717	4.122	6.46	97.1%	53
Multilingual	BPE	$\mathcal{C}_{\text{comp}}$	bottom-up	172,372,100	2.262×10^9	13.122	1.247	4.675	6.96	99.92%	1,483
	TopDownComp	$\mathcal{C}_{\text{comp}}$	top-down	217,983,502	2.678×10^9	12.286	1.266	3.696	5.52	99.94%	618
	BottomUpLL	$\mathcal{C}_{\text{ll}}$	bottom-up	173,917,215	2.245×10^9	12.907	1.347	4.633	7.40	99.36%	1,114
	UnigramLM	$\mathcal{C}_{\text{ll}}$	top-down	240,669,745	2.597×10^9	10.791	1.598	3.348	6.79	99.74%	139

Table 5: Intrinsic statistics in English-only and multilingual settings. All values are computed at 128k vocabulary size. The multilingual evaluation corpus is the union of the five per-language test splits.

	BPE	TopDownComp	BottomUpLL	UnigramLM
BPE	—	37.0%	80.5%	32.3%
TopDownComp	37.0%	—	33.6%	63.6%
BottomUpLL	80.5%	33.6%	—	31.2%
UnigramLM	32.3%	63.6%	31.2%	—

Table 6: Multilingual vocabulary overlap at 128k vocabulary size; the multilingual counterpart of Fig. 3.

decrease, entropies increase, and Coverage 50% grows by an order of magnitude (e.g., BPE: 204 $\rightarrow$ 1,483), reflecting the broader set of frequent token types in a five-language corpus. The relative pattern between objectives is preserved, however: UnigramLM retains the steepest Zipf α and lowest entropy in both settings, and likelihood-based methods stay more concentrated than their compression-based counterparts within each search family.

Vocabulary utilisation rises with corpus breadth. The unused-vocabulary tail visible in English (Vocab Util 96.8–99.8%) largely disappears in the multilingual evaluation (99.36–99.94%), because the broader, more diverse corpus exercises a larger portion of the 128k vocabulary.

Vocabulary overlap follows the same cluster structure. Tab. 6 is the multilingual counterpart of Fig. 3. BPE and BottomUpLL remain the most similar pair (80.5% overlap); TopDownComp and UnigramLM form a second cluster (63.6%); cross-family overlap stays in the 30–37% range.

K Analysis of Merge Dynamics

Merge trajectories are only defined for the bottom-up methods, so this analysis covers BPE and BottomUpLL alone. A token is **absorbed** if it is created by one merge and later appears inside another merge: if a tokeniser first builds *physi* and then merges it into *physic*, *physi* has been absorbed. Absorption is common and not by itself a sign of waste, since an absorbed token usually remains in the vocabulary and can still

Metric	BPE	BottomUpLL (Exact)	BottomUpLL (Approx)
Total absorbed	34,224	41,599	41,751
Absorption rate (all)	26.8%	32.6%	32.7%
Absorption rate ($ s \geq 5$)	21.7%	27.8%	27.9%
Absorption rate ($ s \geq 10$)	4.5%	8.5%	8.8%
Absorbed with zero usage	3.5%	9.2%	8.7%
Absorbed with low usage (≤ 10)	26.2%	36.9%	36.0%
Not absorbed, zero usage	.2%	.4%	.4%

Table 7: Merge-dynamics for bottom-up methods. Low usage means at most 10 occurrences on the test set. Absorbed with zero or low usage are given as a percentage of total absorptions.

used on its own. It only becomes wasteful when the absorbed token reduces in frequency enough to be considered not-useful any more.

Tab. 7 present absorption rates for both BPE and BottomUpLL. Absorption is common for both methods, and by itself says little. The methods separate, however, on the tokens that are absorbed and then never used at all, which BottomUpLL produces nearly three times as often as BPE (9.2% vs. 3.5%). This is where its lower vocabulary utilisation at 128k comes from. Also interestingly, the absorption gap widens for long tokens (8.5% vs. 4.5% at ten characters or more). This suggests BottomUpLL reaches its final vocabulary through more intermediate steps than BPE, which is how it ends up with longer tokens.

L Vocabulary Overlap Structure

Fig. 3 reports pairwise vocabulary overlap in English, but pairwise numbers cannot show how the five vocabularies intersect jointly. Fig. 4 gives the full picture as a five-set Venn diagram over the 128k English-only vocabularies. The structure mirrors the pairwise results: the largest regions are those shared by the two bottom-up methods, the two top-down methods retain the largest exclusive regions, and cross-family regions are comparatively small.

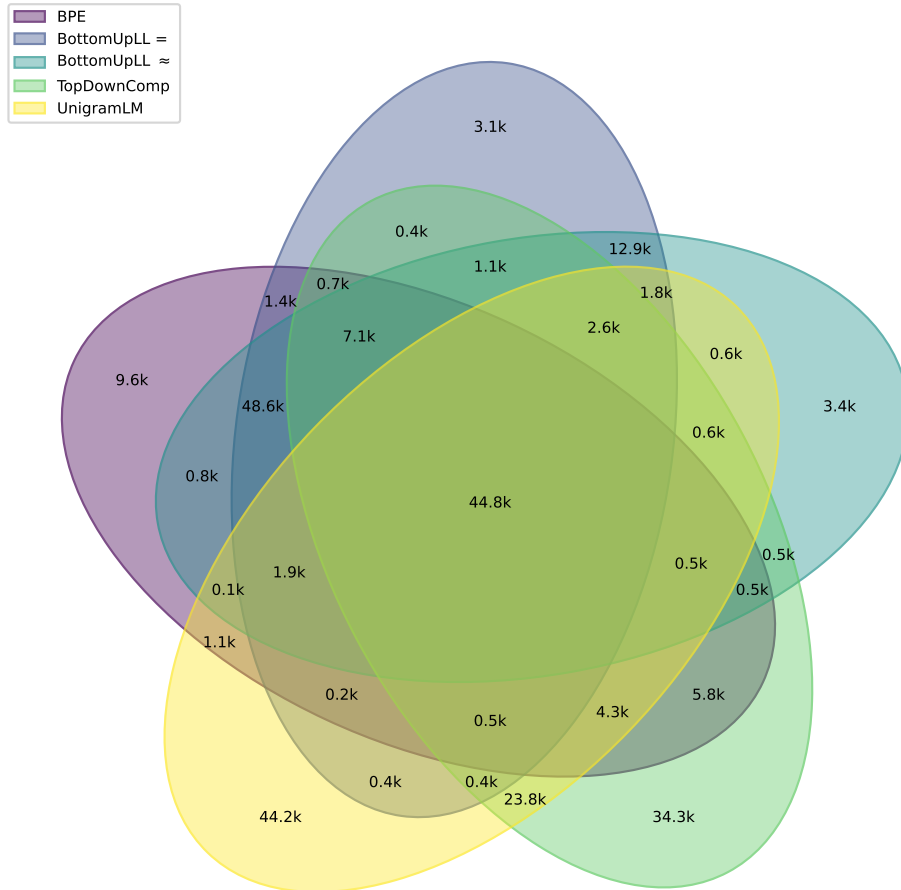


Figure 4: Five-set Venn diagram of the 128k vocabularies. Each region gives the number of tokens (in thousands) belonging to exactly that combination of tokenisers. The largest two-way region is $\text{BPE} \cap \text{BottomUpLL}$, and the BottomUpLL exact and approximate variants are nearly coextensive.

M Qualitative Examples

Aggregate metrics show the dominant trends, but they hide the local segmentation decisions that produce those trends. Here, we inspect a small set of representative examples covering plain English, scientific terminology, code-like text, URLs, Chinese, and multilingual accented text. These examples are illustrative rather than statistically decisive, but they make the inductive biases of the methods easier to interpret. Tab. 8 reports the total number of tokens produced across the 10 illustrative examples. The ordering broadly matches the full compression benchmark.

Scientific and medical terminology. On examples such as:

- (i) Photosynthesis converts carbon dioxide and water into glucose and oxygen.
- or
- (ii) Electroencephalography is used to diagnose neurological disorders.

Method	# of Tokens ↓
BPE	156
TopDownComp	160
BottomUpLL (Exact)	154
BottomUpLL (Approx)	153
UnigramLM	173

Table 8: Total number of tokens produced across 10 illustrative test examples. Lower is better.

BottomUpLL often produces the shortest segmentation. It tends to preserve long technical units such as Photosynthesis or fragments of encephalography as large tokens. BPE is also compact, but more often splits these words into high-frequency pieces. This is consistent with the merge-dynamics analysis: BottomUpLL is more willing to build long lexical chains through intermediate tokens.

Code and punctuation-heavy text. On code-like or SQL-like strings, BPE is usually strongest. For example, on:

(iii) `SELECT * FROM users WHERE id = 1; DROP TABLE users;-`

BPE produces the fewest tokens and preserves punctuation patterns such as `;-`. UnigramLM is much more fragmented, often splitting short code tokens such as `id`, `if`, or `def` into character-level tokens. This reflects the concentration of its token distribution: unless short code tokens are strongly supported by the learned vocabulary, the model falls back to smaller tokens.

URL-like text. For URL strings such as:

(iv) `https://www.example.com/path/to/resource?query=value&page=1`

BPE and TopDownComp are especially compact. BPE often preserves web-specific substrings such as `https`, while BottomUpLL may split them into statistically meaningful but less URL-specific pieces such as `http+s`. This is a case where frequency-driven compression appears better matched to the surface regularities of web text than likelihood-based lexical cohesion.

Chinese text. On the short Chinese example:

(v) 人工智能正在改变世界

UnigramLM produces the fewest tokens among the inspected methods. This does not contradict the multilingual BPB results in Tab. 3, where bottom-up methods are much better on Chinese overall. Rather, it shows that example-level token count and corpus-level language-model BPB can diverge: a tokeniser may preserve some non-Latin chunks well while still producing a worse distribution of tokens for language modelling across the full evaluation set.

Multilingual accented text. Finally, for:

(vi) `iHola! ¿Cómo estás? Très bien, merci. Danke schön!`

all methods produce similar total counts, but the internal boundaries differ. UnigramLM tends to preserve accented fragments such as `Cómo` and `estás` more coherently, while BPE more often splits them into smaller pieces. Thus, even when example-level token counts are similar, tokenisers can impose different boundaries that may matter for downstream behaviour.

Take away. Together, these examples illustrate the same qualitative biases seen in the aggregate analyses: BPE is robust on punctuation-heavy and

web-like text, BottomUpLL tends to form longer lexical units, UnigramLM can preserve some accented or non-Latin fragments while fragmenting other domains, and TopDownComp behaves as a compression-oriented top-down method.

N Significance of BPB Differences

Some BPB differences between tokenisers are small, so we test whether they are robust to the choice of evaluation documents. For each objective-matched pair of tokenisers, we run a paired document-level bootstrap over the held-out English test set (47,384 documents). In each of 10,000 iterations we resample documents with replacement and use the *same* resampled indices for both models, so that the shared per-document difficulty cancels. We then recompute each model’s corpus-level BPB on the resampled set and record the difference between the bottom-up and top-down model. Tab. 9 reports point estimates and 95% confidence intervals for the 128k-vocabulary models, using seed 42 for each configuration.

Size	Comparison	Δ BPB	95% CI
300M	BPE vs. TopDownComp	-0.0031	[-0.0032, -0.0029]
	BottomUpLL vs. UnigramLM	-0.0080	[-0.0082, -0.0078]
500M	BPE vs. TopDownComp	-0.0016	[-0.0017, -0.0014]
	BottomUpLL vs. UnigramLM	-0.0075	[-0.0077, -0.0074]
1B	BPE vs. TopDownComp	-0.0025	[-0.0027, -0.0023]
	BottomUpLL vs. UnigramLM	-0.0039	[-0.0041, -0.0037]

Table 9: Paired document-level bootstrap (10,000 resamples) of BPB differences between objective-matched tokenisers at 128k vocabulary. Negative values indicate the bottom-up method achieves lower BPB. All confidence intervals exclude zero.

All six confidence intervals lie entirely below zero, indicating that, for each objective, the bottom-up tokeniser achieves significantly lower BPB than its top-down counterpart, and that this ordering is not an artifact of the particular held-out documents used for evaluation.

Training-seed variance. The bootstrap above controls for the choice of evaluation documents, but not for randomness in model training. To assess the latter, we retrain the English 1B models at 128k vocabulary with two additional seeds (43 and 44), giving three seeds per tokeniser. Tab. 10 reports the per-seed BPB values. Both objective-matched orderings are preserved in the mean: BPE outperforms TopDownComp, and BottomUpLL outperforms UnigramLM. We note that BottomUpLL

shows a noticeably larger spread than the other tokenisers, driven by a single seed; averaged over three seeds it attains a lower BPB than TopDownComp, whereas on seed 42 alone the ordering is reversed.

Tokeniser	Seed			Mean
	42	43	44	
BPE	0.7795	0.7787	0.7787	$0.7790 \pm 4\text{e-}4$
TopDownComp	0.7819	0.7817	0.7813	$0.7816 \pm 3\text{e-}4$
BottomUpLL	0.7833	0.7789	0.7787	$0.7803 \pm 2\text{e-}4$
UnigramLM	0.7872	0.7871	0.7873	$0.7872 \pm 1\text{e-}4$

Table 10: Per-seed BPB for the English 1B models at 128k vocabulary. Mean is taken over the three seeds, with standard deviation.

O Effect of the Top-Down Approximations

In this section, we investigate the impact of our approximations in top-down methods.

Pruning rate. For efficiency and as is standard practice, our top-down methods remove batches of tokens at each pruning round, rather than a single token. Specifically, our main experiments prune 10% of the current vocabulary per round. To assess how much this batching affects the learned vocabulary, we re-train UnigramLM and TopDownComp at 128k vocabulary using pruning rates of 1% and 0.1% per round, and measure the vocabulary overlap with the corresponding 10% tokeniser. For UnigramLM, the overlap is 97.3% at a 1% pruning rate and 97.1% at 0.1%; for TopDownComp it remains above 99% in both cases. Inspecting the non-overlapping tokens, we find that they occur very rarely in the tokenised corpus. Pruning in batches therefore appears to have a limited effect on the resulting vocabulary.

Local replacement. Our second approximation is the local replacement approximation of §4.2, which scores a deletion by re-segmenting only the deleted token, rather than recomputing the objective over the whole corpus. Computing exact deletion costs requires re-segmenting the corpus once per candidate token at every pruning step, which is intractable at the vocabulary sizes used in our main experiments. We therefore compare the two at a small scale: on 5,000 lines of English FineWeb, we prune from a seed vocabulary of 3,000 tokens down to 800 with a pruning rate of 10%, once using exact

deletion costs and once using the local replacement approximation. The two procedures agree on 652 of the 800 resulting tokens, an overlap of 81.5%. The approximation therefore recovers most, but not all, of the vocabulary that exact scoring would select. We note that this is a small-vocabulary and small-corpus setting, though, and that the effect of the approximation at the scales used in our main experiments remains open.