From Where Words Come: Efficient Regularization of Code Tokenizers Through Source Attribution

Pavel Chizhov[1,2] **Egor Bogomolov**[2,3] **Ivan P. Yamshchikov**[1]

[1]CAIRO, Technical University of Applied Sciences Würzburg-Schweinfurt
[2]JetBrains Research [3]TU Delft

Correspondence: pavel.chizhov@thws.de

Abstract

Efficiency and safety of Large Language Models (LLMs), among other factors, rely on the quality of tokenization. A good tokenizer not only improves inference speed and language understanding but also provides extra defense against jailbreak attacks and lowers the risk of hallucinations. In this work, we investigate the efficiency of code tokenization, in particular from the perspective of data source diversity. We demonstrate that code tokenizers are prone to producing unused, and thus under-trained, tokens due to the imbalance in repository and language diversity in the training data, as well as the dominance of source-specific, repetitive tokens that are often unusable in future inference. By modifying the BPE objective and introducing merge skipping, we implement different techniques under the name Source-Attributed BPE (SA-BPE) to regularize BPE training and minimize overfitting, thereby substantially reducing the number of under-trained tokens while maintaining the same inference procedure as with regular BPE. This provides an effective tool suitable for production use.

pchizhov/sa-bpe

1 Introduction

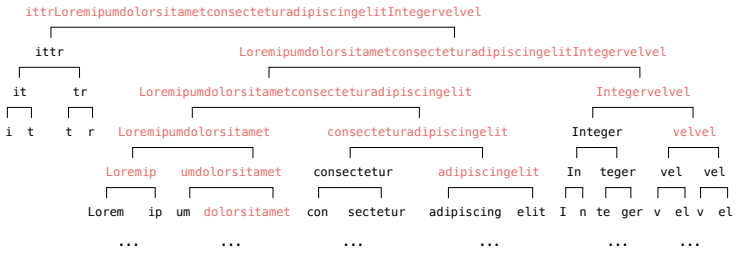
Subword tokenization is the most common approach for vocabulary building in language models (Devlin et al., 2019; Brown et al., 2020; Grattafiori et al., 2024; Kamath et al., 2025). This approach is normally free from linguistic rules and thus fits a variety of arbitrary mixtures of languages, which are commonly not fixed in large models. The main optimization criterion of subword tokenization is text compression. Currently, the most widely used subword tokenization approaches are variants of Byte-Pair Encoding (BPE; Gage, 1994; Sennrich et al., 2016). This algorithm builds a vocabulary by iteratively merging the most frequently co-occurring token pairs, starting from the initial byte sequence. On inference, the learned sequence of merges is applied to a text in the same order as during training.

However, the efficiency of tokenization algorithms comes at the cost of limited control over word segmentation and the resulting vocabulary. One common issue in BPE tokenizers is under-trained tokens (Land and Bartolo, 2024). Such tokens are present in tokenizers but are rarely or never seen during model training. Therefore, these tokens lack adequate embedding representations within the model and merely clutter the vocabulary. Furthermore, because the model is unaware of suitable contexts for these tokens, they can lead to hallucinations or even serve as a means of token-level jailbreak attacks.

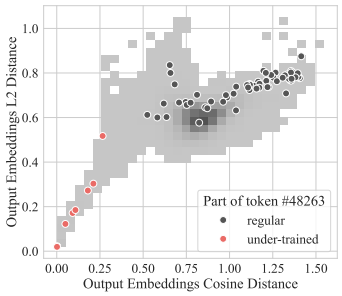
Textual data in coding languages presents a unique challenge for tokenization algorithms. First of all, regardless of the variety of coding languages, English remains the main language for language syntax, while other languages might be present as natural language in comments and docstrings. Second, punctuation and formatting play a large role in code understanding. Finally, variable and function names are frequent and specifically formatted parts of the data: they often contain multiple words without whitespace separation, e.g., using naming conventions such as `CamelCase` and `snake_case`.

To illustrate the problem of under-trained tokens in coding models, we show an outstanding useless token in the StarCoder2 tokenizer (see Figure 1a). The token is a variable name written in `CamelCase`. It comes from a single document that illustrates the maximal variable lengths in Java (Land and Bartolo, 2024). The main content of the token is a placeholder string *"Lorem ipsum dolor..."* written with a typo ("ipum" instead of "ipsum"). Thus, even if another project theoretically included a similar use of this common placeholder phrase, this token or its components would be useless due to the typo. As we highlight in Figure 1, this token

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 39053–39073



(a) The last six levels of the merge tree for token number 48263 in StarCoder2. The token itself is shown at the top of the tree; merges are represented by square brackets.



(b) Under-trained token indicators for the StarCoder2 tokenizer.

Figure 1: Under-trained token example from StarCoder2, token number 48263. We show **(a)** the last steps of the merge tree and **(b)** the under-trained token indicators for StarCoder2 3B computed with the `magikarp` library. The tokens closer to $(0, 0)$ are similar to known under-trained tokens and thus are more likely to be under-trained. Such under-trained tokens that are used to build token number 48263 are highlighted in red in both figures.

is included in the tokenizer along with all its constituent parts, many of which also become under-trained. Other examples include repositories where the names were compressed or randomized and are therefore not informative.

In this work, we investigate the issue of under-trained tokens in code language models and propose **Source-Attributed BPE (SA-BPE)**, a set of BPE modifications to address the problem of overfitting during tokenizer training. We approach this from the perspective of regularization and introduce it in a form opposite to TF-IDF (Salton and McGill, 1983): we de-prioritize terms that appear in only a small subset of documents. In particular, our contributions include:

- We analyze under-trained tokens in the BPE tokenizers across common code models and formulate the main sources of such tokens.
- We propose SA-BPE, several heuristic modifications that introduce regularization into the BPE algorithm and help mitigate overfitting in coding tokenizers.
- We train several small language models and show that our modifications substantially reduce the number of under-trained tokens.

2 Related Work

Tokenization and Regularization. Typical alternatives to BPE include WordPiece and UnigramLM. WordPiece (Schuster and Nakajima, 2012; Wu et al., 2016) is an algorithm for vocabulary learning that iteratively chooses the merges that maximize the overall sequence likelihood the most, which can be reformulated as using the pointwise mutual information (PMI) metric (Church and Hanks, 1990) as an objective. UnigramLM tokenizer (Kudo, 2018), employs a top-down approach, where the large heuristically constructed initial vocabulary is iteratively trimmed by removing tokens that can be split without compromising the sequence probability according to a unigram model. The term "subword regularization" (Kudo, 2018; Provilkov et al., 2020) typically refers to utilizing different tokenization trajectories for the same sequence during language model training and does not imply changes in the BPE vocabulary.

Under-Trained Tokens. Under-trained tokens are primarily a result of overfitting to the tokenizer's training sample, but they can also originate from intermediate tokens that are a natural part of the BPE algorithm and are necessary for inference. Multiple efforts were made to remove junk tokens from language models' vocabularies. Post-training trimming (Yang et al., 2022; Cognetta et al., 2024) focuses on editing the trained BPE tokenizer through removing tokens that are chosen for pruning by their frequency. This method reduces the model size, but requires a task-specific choice of absolute thresholds. Chizhov et al. (2024) and Lian et al. (2025) implemented vocabulary refinement during tokenizer training, mostly targeting intermediate tokens. Both approaches require an inference procedure different from the basic BPE.

BPE Modifications. S-BPE (Vilar and Federico, 2021), similarly to WordPiece, aims at maximizing the overall likelihood of the text. S-BPE includes

Category	Qwen2.5 Coder 3B 2% of 151k tokens	CodeGemma 7B 2% of 262k tokens	StarCoder2 3B 2% of 49k tokens
Special tokens	18 (0.6%)	101 (2.0%)	0 (0.0%)
Digits and numbers	2 (0.1%)	77 (1.5%)	0 (0.0%)
Punctuation	124 (4.1%)	521 (10.2%)	125 (12.2%)
Full or partial variable or function names	4 (0.1%)	64 (1.2%)	451 (44.0%)
ALL–CAPS Latin words	0 (0.0%)	60 (1.2%)	88 (8.6%)
Other Latin words	66 (2.2%)	484 (9.5%)	253 (24.7%)
Non-Latin words or characters	2819 (92.9%)	3120 (60.9%)	27 (2.6%)
Other	0 (0.0%)	693 (13.5%)	80 (7.8%)

Table 1: Under-trained token classification for coding language models. For each under-trained token category, we present the number of tokens and their percentage out of the total studied token set (2% of each vocabulary).

the PMI metric in the objective, accounts for the low-frequency false positives, and adds a stopping criterion based on likelihood decay. Parity-aware BPE (Foroutan et al., 2025), mitigates the issue of language disparity and unfairness on LLM inference (Ahia et al., 2023), by alternating the focus on different languages during tokenizer training, giving favor to less compressed languages.

Other efforts focused on pre-tokenization, which has a large impact on the tokenization that follows. For example, numbers are suggested to be split in groups of one, three left-to-right, or three right-to-left tokens, with the latter having superior performance in the model's numeracy (Lee et al., 2024). Recent work has focused on superword merges, i.e., merges that occur across whitespaces, commonly used as pre-tokenization barriers. Such merges improve text compression and can be useful, especially in coding models, where many instructions are repeated, such as "`import numpy as np`" (Fried et al., 2023). On the other hand, superword tokens might exacerbate overfitting; we investigate this in Section 5.3. Schmidt et al. (2025) and Liu et al. (2026) study the impact of superword merges, introducing them either when pre-tokens are already merged or at a specific stage during BPE training. Schmidt et al. (2025) also note that coding variable names are essentially superword tokens, since they consist of multiple words.

Land and Bartolo (2024) discuss the potential ways to mitigate single-source tokens and suggest introducing an upper bound on per-document pair frequency. We approach this problem from a different perspective, counting the repositories and languages in which each pair appears.

3 Analysis

We study three common coding language models: CodeGemma 7B (Zhao et al., 2024), Qwen 2.5 Coder 3B (Hui et al., 2024), and StarCoder2 3B (Lozhkov et al., 2024) and analyze their under-trained tokens. We chose these because it is straightforward to distinguish their known under-trained tokens: CodeGemma has reserved `<unused>` tokens, Qwen 2.5 Coder has unused embedding vectors above known vocabulary size, and StarCoder2 has a documented set of unused tokens (Land and Bartolo, 2024). For each model, we perform an under-trained token analysis using the `magikarp` library. Using the undertrained token indicators as in Figure 1b, we compute the distance from $(0, 0)$ for each token. Then, we take the lowest 2% by this distance for the analysis. After a manual inspection, we highlight several under-trained token categories. We implement a rule-based classification procedure that we describe in Appendix A and present the results in Table 1.

CodeGemma and Qwen have larger vocabularies, which allows for greater natural language diversity; thus, most of their under-trained tokens are non-Latin words or characters. StarCoder2, in its turn, represents a more condensed tokenizer, where most of the under-trained tokens come from variable names and their parts (44%). Together with punctuation and uppercase Latin tokens, which are typically present in code, they constitute 64.8% of the studied tokens. We argue that such tokens can be efficiently prevented by reducing overfitting to large repositories. Furthermore, since the BPE tokens are derived from frequency maximization, it is highly probable that other categories also originate from single repositories (e.g., project-specific words) or from single corrupted documents. Even the intermediate tokens, the removal of which normally requires changing the tokenization procedure (Chizhov et al., 2024; Lian et al., 2025), can be reduced, since they are often building parts of largely overfitted tokens, as seen in Figure 1a.

4 Methodology

We consider BPE as a greedy optimization algorithm for building a vocabulary $\mathcal{V}$ given a text corpus $\mathcal{C}$. It iteratively collects a list of merges $\mathcal{M}$, enlarging the vocabulary until it reaches the desired size v. For all our experiments, we use the byte-level version of BPE (Radford et al., 2019) that initializes the vocabulary with UTF-8 bytes and does not use special <UNK> tokens.

4.1 BPE Modifications

During BPE training, we track two properties of each pair: $\mathcal{R}$ — the number of repositories where the pair is present, and $\mathcal{L}$ — the number of languages where the pair is present. Using these numbers, we experiment with two BPE modifications that are highlighted in blue in Algorithm 1:

Algorithm 1 BPE Training

Require: Corpus $\mathcal{C}$; desired vocabulary size v
Ensure: Vocabulary $\mathcal{V}$, merge list $\mathcal{M}$

Initialize vocabulary $\mathcal{V}$ from $\mathcal{C}$
$\mathcal{M} \leftarrow [\,]$
while $|\mathcal{V}| < v$ **do**
  $(t_{\text{left}}, t_{\text{right}}) \leftarrow$ PRIORITYCRITERION $(\mathcal{C})$
  if SKIPCRITERION $\left(t_{\text{left}}, t_{\text{right}}, \mathcal{C}\right)$ **then**
    continue
  end if
  $t_{\text{new}} \leftarrow t_{\text{left}} + t_{\text{right}}$
  $\mathcal{V} \leftarrow \mathcal{V} \cup \{t_{\text{new}}\}$
  $\mathcal{M} \leftarrow \mathcal{M} \parallel \left[\left(t_{\text{left}}, t_{\text{right}}\right)\right]$
  Update corpus $\mathcal{C}$ to reflect the merge
end while
return $\mathcal{V}, \mathcal{M}$

Merge skip criterion. We add an option to skip the merges that do not comply with a chosen criterion. For these criteria, we use thresholds $\mathcal{R}_t$ and $\mathcal{L}_t$, meaning that a merge should be present at least in $\mathcal{R}_t$ repositories and $\mathcal{L}_t$ languages to be executed. The intuition behind this approach relies on the fact that these counts do not increase over the merge tree: $\mathcal{R}(t) \leq \mathcal{R}(t_l)$ and $\mathcal{R}(t) \leq \mathcal{R}(t_r)$, where $t = t_l + t_r$; the same is true for the number of languages $\mathcal{L}$. Thus, if a merge is present in k repositories, any merge that the resulting token can form in the future is present in at most k repositories; if k is low, this branch can be pruned. In a sense, increasing the thresholds for merge skipping is a form of adjusting the strength of regularization.

Priority criterion. In the original BPE, pair frequency $\mathcal{F}$ is set as the criterion for choosing the next merge during training. We experiment with adding other components to this objective:

- $\mathcal{F} \cdot \mathcal{L}$, prioritizing language-universal merges.
- $\mathcal{F} \cdot \log(\mathcal{R} + 1)$, lowering the priority of pairs present in fewer repositories.
- $\mathcal{F} \cdot \log \mathcal{R}$, this variant also nullifies all pairs that are present only in one repository.
- $\mathcal{F} \cdot \log(\mathcal{R} + 1) \cdot \mathcal{L}$ and $\mathcal{F} \cdot \log \mathcal{R} \cdot \mathcal{L}$, combining the two properties.

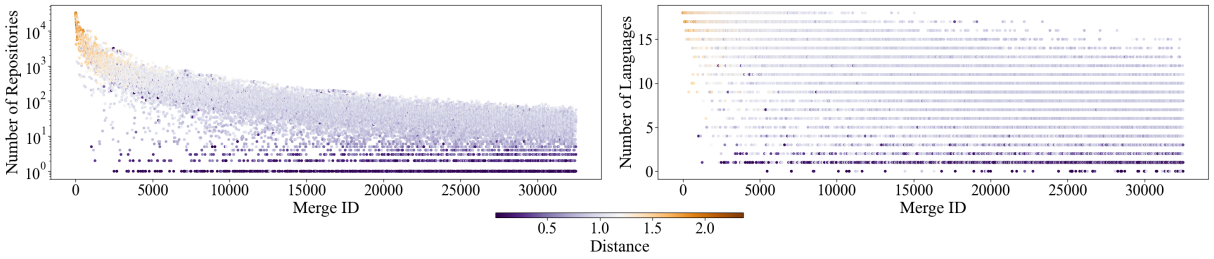
We do not rule out frequencies completely from the criteria, since they are the main component for optimizing text compression, and there would be multiple candidate pairs with equal criterion values without the frequency component. We also experiment with combining merge skip criteria with priority criteria. For example, we add skipping merges where $\mathcal{L} = 1$ to the SA-BPE with the objective $\mathcal{F} \cdot \log \mathcal{R} \cdot \mathcal{L}$, so that all the single-language and single-repository merges are avoided.

Rationale. We propose these modifications to introduce regularization directly into BPE training to prevent overfitting. The key idea behind SA-BPE originates from TF-IDF, which involves dividing by the logarithm of the number of documents to prioritize document-specific terms. Here, however, we want to **de-prioritize** them; therefore, $\log \mathcal{R}$ appears in the numerator. In the code domain, repositories and languages typically have similar structure and syntax, respectively, which makes it natural to use them as units of data.

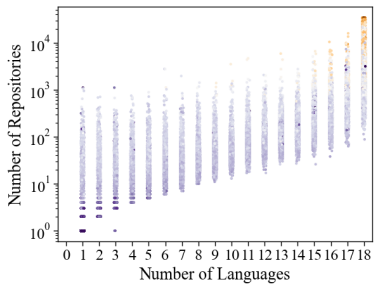
4.2 Experimental setup

We assemble the training dataset using Common Corpus (Langlais et al., 2026). We select a random subset of 7% of documents with a fixed set of 18 programming languages (see full list in Appendix C). The collected sample encompasses 14,428,162 documents and 56.22 GB of text; the per-language statistics are shown in Table 7. For tokenizer training, we select a random subsample of 0.25% of the collected data, comprising 36,052 files. We separately conduct single-language experiments using Java and Python subsets of the assembled data, as high-resource languages that differ in syntax and naming conventions.

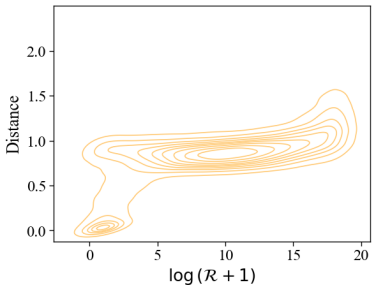
For the evaluation set, we collected a sample



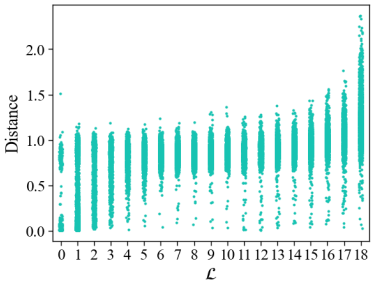
(a) BPE merge history. For each merge, we calculate the number of repositories **(left)** and languages **(right)** that contain this merge. Color represents distance from $(0, 0)$ in the under-trained token identifiers space (lower = more similar to under-trained).



(b) Number of languages vs number of repositories for each merge.



(c) Under-trained token indicator distances vs the number of repositories.



(d) Under-trained token indicator distances vs the number of languages.

Figure 2: BPE merge visualization. We show **(a)** BPE merge sequences with counts of repositories and languages, and **(b)** correspondence between the two counts, highlighting the distances from $(0, 0)$ in the under-trained token indicator space. We also show correlations between these distances and **(c)** repository and **(d)** language counts.

from GitHub repositories that have at least 10 stars and 15 commits, and were created after March 1, 2025, which is later than the cutoff date for Common Corpus, last updated on February 11, 2025 (at the time of this work). For each language, we selected 200 repositories with the most stars; if there were fewer repositories, we included all of them. From each repository, we chose at most 20 files with fixed extensions (see Appendix C for details).

We train a series of small Llama models (Grattafiori et al., 2024) with 100 million parameters (see Appendix B for details). Each model is trained for 60,000 steps on two NVIDIA A100 GPUs with a context window size of 2,048, a batch size per GPU of 16, and four steps of gradient accumulation, resulting in an effective batch size of 262,144 tokens and a total training set size of 15.7 billion tokens. We reserve three unused tokens in each model to analyze the model embeddings with the `magikarp` package (Land and Bartolo, 2024): `<|unused_token_{1,2,3}|>`. These tokens never appear in the training set; hence, their embeddings do not change during training and can be used as seed vectors for under-trained token analysis.

4.3 Evaluation

We compute intrinsic tokenizer quality measures, namely compression rate (the corpus length in bytes divided by its token count), coverage (the number of tokens used at least once in the evaluation sample), and mean token length. We consider tokenizers with better compression and higher coverage on the evaluation set to be more regularized, as they are better suited to unseen content. If a tokenizer with a better compression rate/higher coverage has a lower mean token length, this signals better generalization even further, since shorter types are more likely to be reused, and the long, unneeded tokens take up less space in the vocabulary.

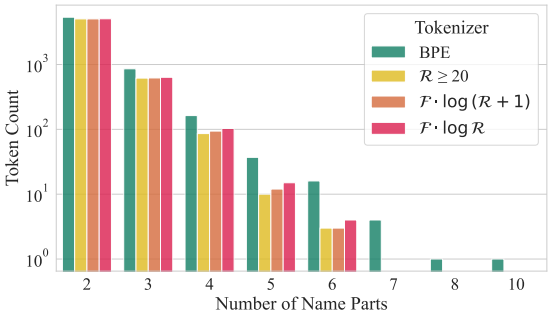
We use the `magikarp` package (Land and Bartolo, 2024) as the main extrinsic measure to study the impact of a tokenizer on a trained model. This evaluation includes under-trained token indicators (Euclidean and cosine distances from the known under-trained tokens) and verification prompts. A verification is run on the bottom 2% of the under-trained token indicator distribution, presenting the model with each token and asking it to repeat it. If the next-token probability for the token itself is below 1%, the token is verified as under-trained.

5 Experimental Results

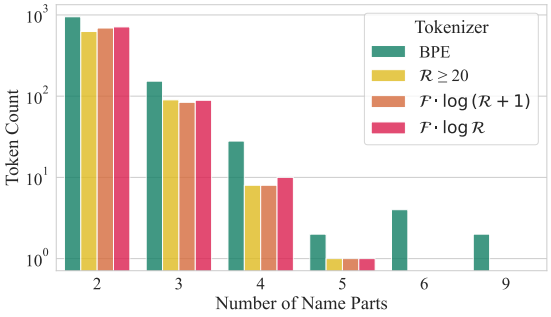
Before implementing our modifications, we train a basic BPE model on the selected tokenizer corpus and count the number of repositories and languages

Tokenizer	CR	Used	MTL	Steps
BPE	3.90	19585	6.75	32508
Skip $\mathcal{R} < 2$	3.90	19862	6.57	36263
Skip $\mathcal{R} < 5$	3.91	20117	6.51	40096
Skip $\mathcal{R} < 10$	3.91	20343	6.50	43575
Skip $\mathcal{R} < 20$	3.92	20741	6.50	51093
$\mathcal{F} \cdot \log(\mathcal{R}+1)$	3.91	20491	6.54	32508
$\mathcal{F} \cdot \log \mathcal{R}$	3.91	20552	6.51	32508
BPE	3.68	17208	5.84	32508
Skip $\mathcal{R} < 2$	3.68	17363	5.78	34186
Skip $\mathcal{R} < 5$	3.69	17493	5.78	36201
Skip $\mathcal{R} < 10$	3.69	17651	5.79	39745
Skip $\mathcal{R} < 20$	3.70	17932	5.84	57570
$\mathcal{F} \cdot \log(\mathcal{R}+1)$	3.69	17670	5.80	32508
$\mathcal{F} \cdot \log \mathcal{R}$	3.69	17708	5.80	32508

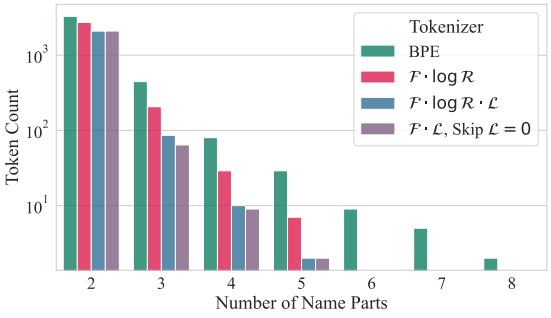
Table 2: Evaluation of Java (top) and Python (bottom) SA-BPE tokenizers compared to BPE: compression rate (CR) and number of used tokens in the evaluation set, mean token length (MTL), and number of training steps.



(a) Variable name token counts in Java tokenizers.



(b) Variable name token counts in Python tokenizers.



(c) Variable name token counts in multilingual tokenizers.

Figure 3: `CamelCase` and `snake_case` token lengths (in name parts) in basic BPE and our modifications.

that each merge appears in. We also train a generative model using this tokenizer and compute under-trained token indicators. We show the BPE merge history in Figure 2a. The number of repositories and languages the merge appears in decreases along the merge history. Furthermore, a certain number of merges occur only in a few repositories and languages, well below the general trend. These merges mostly comprise under-trained tokens. This justifies prioritizing merges that appear across more repositories and languages.

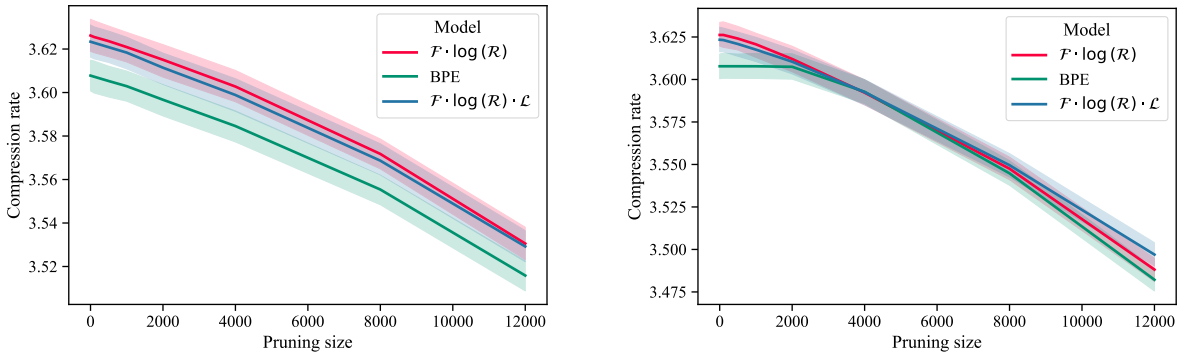
It is also notable that under-trained tokens, positioned low in the plots, appear as early as within the first 5 000 merges. These tokens are, for example: `FHI`, `FHIR`, `PHPFHIR`, `HPFHIRConstants` — tokens appearing in only one PHP project named FHIR, and `chord`, `chordie` — tokens from a repository called `guitartools`. These are clear cases of overfitting to repository-specific naming and content. There are also some under-trained tokens positioned high in the plots. These are mostly intermediate word parts: `perty`, `verride`, `urrent`. These findings correspond well to our hypotheses formulated in Section 3. As we show in Figure 2b, there is a correlation between the number of repositories and the number of languages. Furthermore, Figures 2c and 2d show that the number of repositories is superior to the number of languages in indicating tokens that are clearly under-trained.

5.1 Monolingual Experiments

We train a series of tokenizers on Java and Python samples, using the minimum number of repositories as merge skip criteria and frequencies multiplied by the logarithm of the number of repositories (with and without nullifying single-repository merges) as priority criteria. As we show in Table 2, SA-BPE modifications slightly improve compression rate and token coverage in unseen repositories, both of which tend to increase along with the strength of regularization (increasing the threshold for $\mathcal{R}$). Compared to BPE, our tokenizers have a lower mean token length, which might serve as a proxy measure for overfitting to longer repository-specific names and, together with the non-compromised compression rate, suggests better generalization. While increasing the threshold for merge skips improves the tokenizer's generalizability, it comes with longer training, requiring

Priority Criterion	Merge Skip	Compression	Coverage	MTL	# 3-digit numbers	# Under-trained
$\mathcal{F}$ (BPE)	— (BPE)	3.623	29240	6.23	517	640
$\mathcal{F}$	Skip $\mathcal{L} < 4$	3.623	31416	5.89	655	52
$\mathcal{F} \cdot \mathcal{L}$	—	3.637	31267	5.85	772	151
$\mathcal{F} \cdot \mathcal{L}$	Skip $\mathcal{L} = 1$	3.635	31357	5.84	779	75
$\mathcal{F} \cdot \log(\mathcal{R}+1)$	—	3.641	30876	6.02	595	202
$\mathcal{F} \cdot \log \mathcal{R}$	—	3.643	31066	5.99	607	72
$\mathcal{F} \cdot \log(\mathcal{R}+1)$	Skip $\mathcal{L} < 4$	3.628	31187	5.89	608	62
$\mathcal{F} \cdot \log(\mathcal{R}+1) \cdot \mathcal{L}$	—	3.642	31533	5.86	800	63
$\mathcal{F} \cdot \log \mathcal{R} \cdot \mathcal{L}$	—	3.642	31567	5.86	804	44
$\mathcal{F} \cdot \log \mathcal{R} \cdot \mathcal{L}$	Skip $\mathcal{L} = 1$	3.639	31581	5.86	808	45

Table 3: Evaluation of multilingual SA-BPE tokenizers compared to BPE: compression rate and number of total used tokens in the evaluation set, mean token length (MTL), three-digit numbers count, and number of verified under-trained tokens out of 646 tested. The first row represents the basic BPE algorithm.



(a) Pruning order: reverse merge order.

(b) Pruning order: under-trained first.

Figure 4: Compression rate for tokenizers with applied pruning **(a)** in the reverse order of token ids and **(b)** starting from the tokens with the lowest under-trained indicator values, here distance from $(0, 0)$ in indicator space.

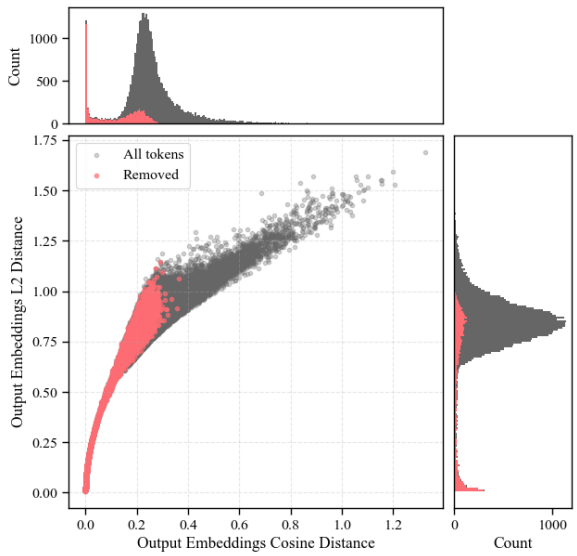
more steps due to skips. Furthermore, increasing the threshold to 30 already resulted in running out of merges for the vocabulary of size 32 768.

In addition, we investigate the presence of `CamelCase` and `snake_case` tokens in tokenizers. For each such token, we compute its length in name parts, splitting the `snake_case` by underscores and `CamelCase` by capital letters (except for the abbreviations, e.g., `getHTTPStatus` has three parts) and show the distributions in Figures 3a and 3b. Compared to BPE, our tokenizers have fewer long variable names. In all evaluations, priority criteria based on the logarithm of repository count showed comparable stable performance, without requiring a threshold value. Out of the two priority criterion versions, $\mathcal{F} \cdot \log \mathcal{R}$ shows slightly better scores as it avoids single-repository tokens.

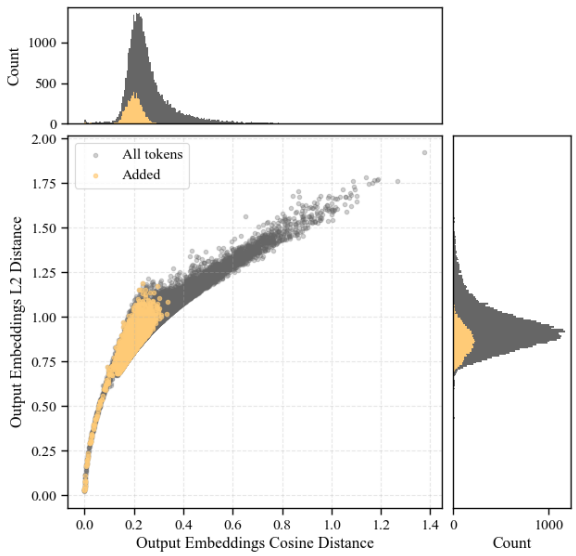
5.2 Multilingual Experiments

Tokenizer metrics. In multilingual experiments, we incorporate the language component into the analysis as a multiplier or a threshold, choosing an intermediate value of 4 (we compare different threshold values in Appendix D). We present the main tokenizer metrics in Table 3. Most of the tested SA-BPE modifications improve text compression, increase the coverage of the tokenized set, and yield generally shorter tokens than the basic BPE. Per-language scores indicate that modifications incorporating the language component provide stronger support for lower-resource languages (see Appendix G for a comparison). Our regularized versions also produce more three-digit numbers as tokens, closer to 900, the total count in the best case. Comparing the lengths of variable names in Figure 3c, we observe a situation similar to the monolingual scenarios: regularized tokenizers have fewer tokens that are long variable names; in particular, no longer than five name parts.

We separately evaluate compression during tokenizer pruning, using two pruning strategies: starting from the end of the merge list (Figure 4a) and leaf-based pruning (Purason et al., 2026) starting from the most under-trained tokens (Figure 4b). In



(a) Under-trained token indicators for the model trained with basic BPE. The tokens present in this model but not in the $\mathcal{F} \cdot \log \mathcal{R} \cdot \mathcal{L}$ model are highlighted in color.



(b) Under-trained token indicators for the $\mathcal{F} \cdot \log \mathcal{R} \cdot \mathcal{L}$ model. The tokens present in this model but not in the model trained with basic BPE are highlighted in color.

Figure 5: Under-trained token analysis in comparison for the model trained with basic BPE and the model trained with priority criterion $\mathcal{F} \cdot \log \mathcal{R} \cdot \mathcal{L}$. Each plot is accompanied by 1D histograms for better visual comparison.

the first case, we observe that our modifications result in superior compression across all vocabulary sizes. In the second case, the first 2,000 removed tokens have no influence on BPE's compression rate. This confirms that BPE produces useless tokens and that these tokens are well-correlated with the under-trained token indicators from `magikarp`. At the same time, our regularized tokenizers begin to lose compression even with as few as 100 removed tokens, highlighting the low redundancy in the vocabulary. We present and analyze the per-language plots in Appendix I.

In Figure 6, we show token probability distributions in the training sample. BPE shows a heavy tail in the lowest-probability zone, which signals that more tokens will receive insufficient training.

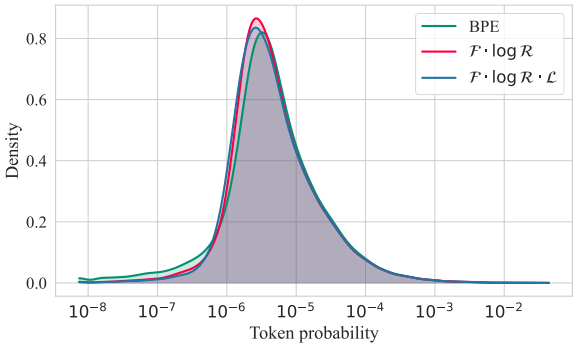


Figure 6: Kernel density estimation of the token probabilities calculated on the model training data.

Model evaluations. The critical part of our analysis is the quantity of under-trained tokens. We used the verification prompting procedure with default parameters from `magikarp` and cosine distance as the indicator metric. For all tokenizers, the total number of tested tokens was 646. All SA-BPE variants reduce the number of verified under-trained tokens, in the best case, down to 44. Repository and language counts are efficient on their own, but combined modifications showed the best performance, especially as parts of the priority criterion. The difference is also visible on the under-trained token indicator plots (see Figure 5). The histograms of the basic BPE show large bins near $(0, 0)$, indicating under-trained tokens. In the modified algorithm, these regions are rather negligible. These tokens are unavoidable intermediate parts of frequent words, e.g., `OOLEAN`, `scriptors`, `trieve` — parts of `BOOLEAN`, `descriptors`, `retrieve` (see examples from all models in Appendix H). To avoid such tokens, it is possible to use SA-BPE regularization in the algorithms aimed at intermediate tokens, e.g., PickyBPE. Examining the removed and added tokens in the same figure, we observe that SA-BPE excludes the majority of under-trained tokens from BPE and includes tokens closer to the general distribution. The difference in this case exceeds 10,000 tokens (see Appendix G for all differences).

Tokenizer	CR	Used	MTL	Time
BPE	3.62	29240	6.23	**0:12**
PickyBPE	3.61	28673	6.22	27:04*
BoundlessBPE	3.56	29375	6.70	28:55*
SA-BPE (ours)	**3.64**	**31567**	5.86	**0:12**

Table 4: Evaluation of multilingual tokenizers: compression rate (CR) and number of used tokens in the evaluation set, mean token length (MTL), and total inference time. "SA-BPE" denotes our best model trained with priority criterion $\mathcal{F} \cdot \log \mathcal{R} \cdot \mathcal{L}$. *Inference time is implementation-dependent.

5.3 Alternative Tokenization Methods

We also test the performance of UnigramLM and Wordpiece[1] on code data in Appendix F and find those to be inferior even to basic BPE. We also compare our regularized BPE tokenizers to common BPE modifications: BoundlessBPE (Schmidt et al., 2025) and PickyBPE (Chizhov et al., 2024). Our best method, SA-BPE with a priority criterion $\mathcal{F} \cdot \log \mathcal{R} \cdot \mathcal{L}$, achieves the best compression rate and token coverage. Along with these scores, our method has a lower mean token length, which, combined with a better compression rate, signals less overfitting. Finally, because the inference procedure remains intact, our method demonstrates superior inference time[2].

6 Discussion

The demonstrated SA-BPE variants introduce regularization into BPE training and demonstrate superiority in building a generalizable tokenizer for coding languages in both monolingual and multilingual scenarios, compared to both basic BPE and SOTA alternatives. Though results suggest that the repository count is a more informative metric and language count might be its proxy (Figure 2b), the models with the fewest under-trained tokens and the best performance include both components. This might be due to supporting the general language-agnostic syntax tokens in the multilingual environment. Also, language-related modifications show better parity and support for lower-resource languages (Appendix G). Merge skip criteria are explicit, but require parameter tuning. Therefore, the more obvious cases for them are prohibiting single-language tokens in certain multilingual scenarios and setting explicit lower bounds when the project knowledge allows. Based on these findings, one might choose modifications depending on project needs, while the combined parameter-free priority criterion modifications ($\mathcal{F} \cdot \log \mathcal{R} \cdot \mathcal{L}$ in the multilingual case) are a good default choice.

One of the benefits of SA-BPE is that it brings changes only to the BPE training, while the tokenizer can be used for inference just as the regular BPE (see the discussion of time and space complexity in Appendix E). The demonstrated approach can also be applied to natural languages, using analogies of data sources such as documents, books, or document collections. However, the language aspect must be avoided in this case, since language diversity in natural domains is substantially larger than in programming languages.

7 Conclusion

In this paper, we propose SA-BPE, a set of modifications to the BPE training procedure that enable a substantial reduction in overfitting for code data, without altering the tokenization function and slowing down inference. Our modifications effectively target repository-specific names and variables, as well as the non-code-related content. This improves tokenizer efficiency and reduces the number of under-trained tokens, which are linked with various model issues in the literature. Algorithmically, SA-BPE is lightweight, which makes it possible to combine it with other BPE modifications and apply it to other domains and general natural language, using the analogy of data sources.

Limitations

In this work, we focus on the code data as the primary domain and utilize its specific characteristics to propose improvements. Even though we mention in Section 6 that the proposed modifications are, in theory, applicable to other domains, we do not conduct such experiments, since they are out of scope of the current work.

Due to computational resource limitations, the models trained in this work are small and not suitable for more sophisticated benchmarking with downstream code-related tasks. For the same reason, we also did not study how the proposed changes would affect the performance of larger language models with vocabulary transfer and continued pre-training on the code data.

[1]We use the `tokenizers` library from HuggingFace, where WordPiece is implemented as BPE with a greedy longest-prefix inference.

[2]For both alternatives, we use byte-level implementations from https://github.com/kensho-technologies/boundlessbpe

Acknowledgements

This work was funded by a joint project of the Center for Artificial Intelligence (CAIRO), THWS and JetBrains Research (project ERIC). The computational resources were provided by Erlangen National High-Performance Computing Center (NHR@FAU) of the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) (joint project with CAIRO, THWS). The authors acknowledge the work of Taido Purason, specifically the repository `tokenizer-extension`, the BPE training implementation from which was adapted for the core implementation of this work.

References

Orevaoghene Ahia, Sachin Kumar, Hila Gonen, Jungo Kasai, David Mortensen, Noah Smith, and Yulia Tsvetkov. 2023. Do All Languages Cost the Same? Tokenization in the Era of Commercial Language Models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 9904–9923, Singapore. Association for Computational Linguistics.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, and 12 others. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

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.

Kenneth Ward Church and Patrick Hanks. 1990. Word Association Norms, Mutual Information, and Lexicography. *Computational Linguistics*, 16(1):22–29.

Marco Cognetta, Tatsuya Hiraoka, Rico Sennrich, Yuval Pinter, and Naoaki Okazaki. 2024. An Analysis of BPE Vocabulary Trimming in Neural Machine Translation. In *Proceedings of the Fifth Workshop on Insights from Negative Results in NLP*, pages 48–50, Mexico City, Mexico. Association for Computational Linguistics.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *North American Chapter of the Association for Computational Linguistics*.

Negar Foroutan, Clara Meister, Debjit Paul, Joel Niklaus, Sina Ahmadi, Antoine Bosselut, and Rico Sennrich. 2025. Parity-Aware Byte-Pair Encoding: Improving Cross-lingual Fairness in Tokenization. *Preprint*, arXiv:2508.04796.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations*.

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

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, and 542 others. 2024. The Llama 3 Herd of Models. *Preprint*, arXiv:2407.21783.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. Qwen2.5-Coder Technical Report. *Preprint*, arXiv:2409.12186.

Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, Thomas Mesnard, Geoffrey Cideron, Jean bastien Grill, Sabela Ramos, Edouard Yvinec, Michelle Casbon, Etienne Pot, Ivo Penchev, Gaël Liu, and 196 others. 2025. Gemma 3 Technical Report. *Preprint*, arXiv:2503.19786.

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, Melbourne, Australia. Association for Computational Linguistics.

Sander Land and Max Bartolo. 2024. Fishing for Magikarp: Automatically Detecting Under-trained Tokens in Large Language Models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 11631–11646, Miami, Florida, USA. Association for Computational Linguistics.

Pierre-Carl Langlais, Pavel Chizhov, Catherine Arnett, Carlos Rosas Hinostroza, Mattia Nee, Eliot Krzysztof Jones, Irène Girard, David Mach, Anastasia Stasenko, and Ivan P. Yamshchikov. 2026. Common Corpus: The Largest Collection of Ethical Data for LLM Pre-Training. In *The Fourteenth International Conference on Learning Representations*.

Garreth Lee, Guilherme Penedo, Leandro von Werra, and Thomas Wolf. 2024. From Digits to Decisions: How Tokenization Impacts Arithmetic in LLMs.

Haoran Lian, Yizhe Xiong, Jianwei Niu, Shasha Mo, Zhenpeng Su, Zijia Lin, Hui Chen, Jungong Han, and Guiguang Ding. 2025. Scaffold-BPE: Enhancing Byte Pair Encoding for Large Language Models with Simple and Effective Scaffold Token Removal. In *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence*, AAAI'25/IAAI'25/EAAI'25. AAAI Press.

Alisa Liu, Jonathan Hayase, Valentin Hofmann, Sewoong Oh, Noah A. Smith, and Yejin Choi. 2026. SuperBPE: Space Travel for Language Models. In *Tokenization Workshop*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, and 47 others. 2024. StarCoder 2 and The Stack v2: The Next Generation. *Preprint*, arXiv:2402.19173.

Ivan Provilkov, Dmitrii Emelianenko, and Elena Voita. 2020. BPE-Dropout: Simple and Effective Subword Regularization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1882–1892, Online. Association for Computational Linguistics.

Taido Purason, Pavel Chizhov, Ivan P. Yamshchikov, and Mark Fishel. 2026. Teaching old tokenizers new words: Efficient tokenizer adaptation for pretrained models. In *Findings of the Association for Computational Linguistics: EACL 2026*, pages 6492–6516, Rabat, Morocco. Association for Computational Linguistics.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners.

Gerard Salton and Michael McGill. 1983. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, NY.

Craig W Schmidt, Varshini Reddy, Chris Tanner, and Yuval Pinter. 2025. Boundless byte pair encoding: Breaking the pre-tokenization barrier. In *Second Conference on Language Modeling*.

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, Berlin, Germany. Association for Computational Linguistics.

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, Bangkok, Thailand (online). Association for Computational Linguistics.

Yonghui Wu, Mike Schuster, Z. Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz 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*, abs/1609.08144.

Ziqing Yang, Yiming Cui, and Zhigang Chen. 2022. TextPruner: A Model Pruning Toolkit for Pre-Trained Language Models. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 35–43, Dublin, Ireland. Association for Computational Linguistics.

Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A. Choquette-Choo, Jingyue Shen, Joe Kelley, Kshitij Bansal, Luke Vilnis, Mateo Wirth, Paul Michel, Peter Choy, Pratik Joshi, Ravin Kumar, Sarmad Hashmi, Shubham Agrawal, Zhitao Gong, and 7 others. 2024. CodeGemma: Open Code Models Based on Gemma. *Preprint*, arXiv:2406.11409.

A Rule-Based Classification

To classify the tokens, we use the following rules and regular expressions:

- **Special tokens:** tokens fitting the regular expression `r"<[^>]+>"`.
- **Digits and numbers:** digit sequences, optionally preceeded by a "+" or a "-" sign.
- **Punctuation:** tokens where symbols that are present in `string.punctuation` or that have a Unicode category that assumes punctuation constitute more than 80%.
- **Variable or function names:** tokens that consist of words merged via underscores ("_") or matching the regular expression `r"^[a-zA-Z]*[A-Z][a-zA-Z]*$"`.
- **ALL–CAPS Latin words:** tokens consisting of Latin letters, all in upper case.

- **Other Latin words:** tokens consisting of Latin letters that did not classify into previous categories.
- **Non-Latin words or characters:** tokens that contain non-Latin characters.
- **Other:** tokens not falling into any category.

B Model and Training

In Tables 5 and 6, we show the main model and training parameters, respectively.

Model Parameter	Value
Hidden size	768
Intermediate size	2048
Number of hidden layers	12
Number of attention heads	12
Number of key-value heads	4
Max position embeddings	2048
Vocabulary size	32768
Hidden activation	`silu`
Initializer range	0.02
Tie word embeddings	`true`
RMS norm epsilon	1.0×10^{-5}
ROPE theta	10000

Table 5: Main model configuration parameters

Training Parameter	Value
Optimizer	`adamW`
Weight decay	0.01
Gradient clipping	1.0
Adam beta1	0.9
Adam beta2	0.95
Adam epsilon	1.0×10^{-8}
Warmup steps	5000
Total steps	60000
Learning rate decay style	`linear`
Learning rate warmup style	`linear`
Max learning rate	0.0003
End learning rate	0.000003

Table 6: Main training configuration parameters

C Data

In Tables 7 and 8, we show language statistics for the training corpus and the extensions chosen from the repositories for each language in the process of assembling the evaluation dataset, respectively.

Language	# Documents	Size (in GB)
Java	2,741,833	11.15
JavaScript	2,757,358	8.40
Python	1,770,216	7.90
PHP	2,006,486	7.02
C++	990,971	6.67
C#	1,468,542	5.70
Go	615,934	2.81
Rust	185,878	1.13
Ruby	589,494	1.07
Kotlin	320,726	0.78
Scala	242,087	0.76
Swift	234,583	0.76
Vue	182,438	0.75
Dart	116,085	0.42
Lua	71,166	0.33
Haskell	73,609	0.28
Julia	37,525	0.16
OCaml	23,231	0.14

Table 7: Per-language dataset statistics

Language	Extensions
Java	`java`
C#	`cs`
C++	`cpp`, `hpp`
Python	`py`
Haskell	`hs`
Dart	`dart`
Go	`go`
JavaScript	`js`
Julia	`jl`
Kotlin	`kt`
Ruby	`rb`
Rust	`rs`
Scala	`scala`, `sc`
Swift	`swift`
Vue	`vue`
PHP	`php`
Lua	`lua`
OCaml	`ml`, `mli`

Table 8: File extensions chosen for each language for the evaluation set.

D Language Skip Criterion

We separately investigate different numbers of languages $\mathcal{L}$ chosen as skip criteria. Our results in Figure 7 show that with the increase of the threshold, for some languages (mainly higher-resource ones: C#, Java, C++), the compression tends to decrease, while slightly increasing for others (mainly low-resource ones: Lua, Dart, Vue). This corresponds well to findings of Foroutan et al. (2025), where the authors also observe a slight decrease in overall compression rate while improving it for lower-resource languages. For most of the languages, the increase in threshold is accompanied by improved coverage.

In addition, in Table 9, we show that there is a tradeoff between compression rate and gini index for the compression rates per language.

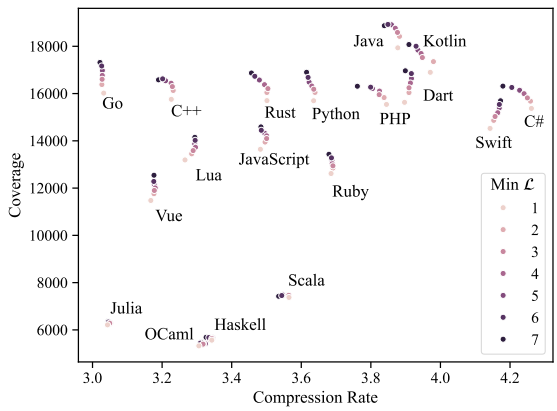


Figure 7: Models with thresholds for the minimum number of languages evaluated for compression rate and coverage. Minimum $\mathcal{L} = 1$ is regular BPE. Tokenizers with $\mathcal{L} > 7$ resulted in exhausting the merge queue.

E Time and Space Complexity

Training. Out of the two modifications, only the merge skip criterion increases the number of main BPE loop steps. However, the most computation-intensive algorithm parts are related to post-merge frequency and corpus updates, which do not happen if a merge is skipped. Therefore, this part potentially adds a new component with an upper bound of $\mathcal{O}(|Q|)$ to the time complexity, where $|Q|$ is the size of the priority queue. This does not change the overall time complexity as $\mathcal{O}(|Q|)$ is already the complexity of the queue initialization.

The main difference from the original BPE, however, lies in memory consumption, as our modifications require separate structures to maintain the mappings of each pair to its repositories and languages, along with the separate frequency counters. Therefore, the memory consumption increases by $\mathcal{O}(P \cdot (\mathcal{R}_{max} + \mathcal{L}_{max}))$, where P is the number of pairs in the queue. The updates in these structures also scale the merge time complexity during training comparably to the frequency-based updates, based on the number of repositories and/or languages in which the pair is present.

Inference. The inference stage of all our modifications is identical to that of the basic BPE, as every modification only affects the order of merges and the resulting vocabulary. Thus, we do not bring any overhead to the inference procedure, in terms of both memory and time. Instead, having a more optimal vocabulary leads to a slight increase in inference speed due to improved vocabulary utilization and compression.

Min $\mathcal{L}$	Gini	Compression
1 (BPE)	0.0576	3.5693
2	0.0576	3.5743
3	0.0570	3.5719
4	0.0568	3.5710
5	0.0567	3.5651
6	0.0566	3.5588
7	0.0559	3.5498

Table 9: Gini index and compression rate for tokenizers regularized with language-based merge skip criteria.

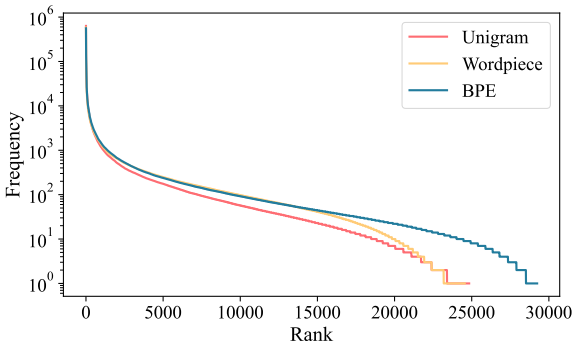


Figure 8: Token frequencies distribution in the evaluation corpus for basic BPE, Unigram, and Wordpiece tokenizers. Total coverage for BPE is 89.23%, for Unigram 75.79%, and for Wordpiece 74.92%.

F Unigram and Wordpiece

We separately train Wordpiece and Unigram tokenizers on the same multilingual data sample and compare coverage and frequency distributions in Figure 8. BPE shows superior coverage, and its frequency distribution is substantially higher than that of the other tokenizers. By manual inspection, the Unigram tokenizer contains more than 5000 tokens as single Chinese characters or emojis, while the Wordpiece tokenizer contains many overly long variable names, e.g., "`testTypeConstantsDefinedRistekUSDI_FHIR_R4B_FHIRResource_FHIR`" as a single token. Based on these results, we excluded these methods from the analysis and compared only the BPE-based algorithms.

G Multilingual Models

In Tables 10 and 11, we present the compression rates by language for BPE and our modified models. In Tables 12 and 13, we present language coverage in the same manner. Modifications that include a language component lead to better per-language scores. This effect becomes more pronounced in lower-resource languages, suggesting that these

Priority Criterion	Merge Skip	Java	JavaScript	Python	PHP	C++	C#	Go	Rust	Ruby
$\mathcal{F}$ (BPE)	— (BPE)	3.875	3.481	3.634	3.842	3.225	4.265	3.030	3.499	3.683
$\mathcal{F}$	Skip $\mathcal{L} < 4$	3.867	3.499	3.629	3.822	3.224	4.244	3.026	3.492	3.686
$\mathcal{F} \cdot \mathcal{L}$	—	3.887	3.507	3.644	3.848	3.238	4.267	3.037	3.507	3.692
$\mathcal{F} \cdot \mathcal{L}$	Skip $\mathcal{L} = 1$	3.880	3.508	3.643	3.837	3.235	4.263	3.030	3.506	3.693
$\mathcal{F} \cdot \log(\mathcal{R}+1)$	—	3.901	3.505	3.650	3.859	3.239	4.284	3.040	3.513	3.702
$\mathcal{F} \cdot \log \mathcal{R}$	—	3.903	3.507	3.652	3.860	3.240	4.286	3.041	3.514	3.703
$\mathcal{F} \cdot \log(\mathcal{R}+1)$	Skip $\mathcal{L} < 4$	3.875	3.507	3.633	3.826	3.226	4.249	3.029	3.494	3.691
$\mathcal{F} \cdot \log(\mathcal{R}+1) \cdot \mathcal{L}$	—	3.896	3.514	3.651	3.853	3.239	4.275	3.040	3.511	3.695
$\mathcal{F} \cdot \log \mathcal{R} \cdot \mathcal{L}$	—	3.896	3.515	3.651	3.854	3.239	4.276	3.040	3.511	3.696
$\mathcal{F} \cdot \log \mathcal{R} \cdot \mathcal{L}$	Skip $\mathcal{L} = 1$	3.890	3.515	3.650	3.842	3.236	4.270	3.033	3.508	3.696

Table 10: Text compression by language for the nine higher-resource languages.

Priority Criterion	Merge Skip	Kotlin	Scala	Swift	Vue	Dart	Lua	Haskell	Julia	OCaml
$\mathcal{F}$ (BPE)	— (BPE)	3.968	3.562	4.140	3.165	3.894	3.264	3.340	3.041	3.305
$\mathcal{F}$	Skip $\mathcal{L} < 4$	3.941	3.559	4.162	3.178	3.911	3.294	3.340	3.049	3.319
$\mathcal{F} \cdot \mathcal{L}$	—	3.980	3.567	4.167	3.186	3.912	3.296	3.350	3.055	3.334
$\mathcal{F} \cdot \mathcal{L}$	Skip $\mathcal{L} = 1$	3.980	3.564	4.166	3.187	3.912	3.296	3.342	3.056	3.335
$\mathcal{F} \cdot \log(\mathcal{R}+1)$	—	3.985	3.574	4.162	3.181	3.920	3.284	3.356	3.052	3.318
$\mathcal{F} \cdot \log \mathcal{R}$	—	3.987	3.576	4.166	3.183	3.921	3.285	3.357	3.055	3.316
$\mathcal{F} \cdot \log(\mathcal{R}+1)$	Skip $\mathcal{L} < 4$	3.948	3.563	4.167	3.182	3.918	3.294	3.344	3.055	3.322
$\mathcal{F} \cdot \log(\mathcal{R}+1) \cdot \mathcal{L}$	—	3.988	3.567	4.171	3.189	3.919	3.296	3.364	3.054	3.331
$\mathcal{F} \cdot \log \mathcal{R} \cdot \mathcal{L}$	—	3.989	3.567	4.171	3.190	3.919	3.296	3.363	3.055	3.332
$\mathcal{F} \cdot \log \mathcal{R} \cdot \mathcal{L}$	Skip $\mathcal{L} = 1$	3.989	3.564	4.169	3.190	3.920	3.297	3.345	3.054	3.332

Table 11: Text compression by language for the nine lower-resource languages.

Priority Criterion	Merge Skip	Java	JavaScript	Python	PHP	C++	C#	Go	Rust	Ruby
$\mathcal{F}$ (BPE)	— (BPE)	17935	13643	15693	15541	15755	15370	16019	15701	12618
$\mathcal{F}$	Skip $\mathcal{L} < 4$	18763	14211	16315	16106	16437	15993	16767	16376	13041
$\mathcal{F} \cdot \mathcal{L}$	—	18816	14359	16541	16266	16549	16163	16993	16594	13251
$\mathcal{F} \cdot \mathcal{L}$	Skip $\mathcal{L} = 1$	18850	14377	16561	16278	16566	16188	17007	16625	13267
$\mathcal{F} \cdot \log(\mathcal{R}+1)$	—	18688	14130	16247	16113	16279	15890	16574	16180	12965
$\mathcal{F} \cdot \log \mathcal{R}$	—	18756	14184	16289	16160	16314	15926	16627	16231	13000
$\mathcal{F} \cdot \log(\mathcal{R}+1)$	Skip $\mathcal{L} < 4$	18990	14386	16497	16314	16548	16140	16962	16530	13181
$\mathcal{F} \cdot \log(\mathcal{R}+1) \cdot \mathcal{L}$	—	18991	14471	16655	16397	16614	16238	17079	16636	13317
$\mathcal{F} \cdot \log \mathcal{R} \cdot \mathcal{L}$	—	19004	14481	16663	16413	16625	16250	17087	16635	13321
$\mathcal{F} \cdot \log \mathcal{R} \cdot \mathcal{L}$	Skip $\mathcal{L} = 1$	19030	14499	16669	16404	16635	16265	17103	16644	13331

Table 12: Language coverage for the nine higher-resource languages.

Priority Criterion	Merge Skip	Kotlin	Scala	Swift	Vue	Dart	Lua	Haskell	Julia	OCaml
$\mathcal{F}$ (BPE)	— (BPE)	16898	7375	14528	11475	15626	13189	5565	6214	5325
$\mathcal{F}$	Skip $\mathcal{L} < 4$	17693	7475	15191	12006	16443	13726	5653	6296	5405
$\mathcal{F} \cdot \mathcal{L}$	—	17834	7484	15353	12262	16649	13875	5650	6354	5431
$\mathcal{F} \cdot \mathcal{L}$	Skip $\mathcal{L} = 1$	17871	7483	15380	12295	16681	13908	5654	6360	5434
$\mathcal{F} \cdot \log(\mathcal{R}+1)$	—	17561	7460	15005	11924	16229	13568	5610	6293	5381
$\mathcal{F} \cdot \log \mathcal{R}$	—	17627	7473	15053	11975	16291	13608	5617	6298	5387
$\mathcal{F} \cdot \log(\mathcal{R}+1)$	Skip $\mathcal{L} < 4$	17940	7482	15359	12203	16685	13866	5634	6326	5417
$\mathcal{F} \cdot \log(\mathcal{R}+1) \cdot \mathcal{L}$	—	17979	7476	15413	12374	16742	13940	5650	6357	5432
$\mathcal{F} \cdot \log \mathcal{R} \cdot \mathcal{L}$	—	17987	7479	15422	12385	16760	13948	5650	6359	5431
$\mathcal{F} \cdot \log \mathcal{R} \cdot \mathcal{L}$	Skip $\mathcal{L} = 1$	18011	7481	15435	12407	16781	13963	5649	6362	5432

Table 13: Language coverage for the nine lower-resource languages.

modifications provide better support for them. This is most evident in compression rates for modifications that include $\mathcal{L} < 4$: for higher-resource languages, these models exhibit inferior compression, whereas they achieve better scores for most lower-resource languages. We hypothesize that this modification leads to stronger regularization of the language component.

In Table 14, we present a pairwise vocabulary comparison of tokenizers. The tokenizers that combine both language and repository components show the largest difference from BPE, with more than 10,000 tokens. Similar tokenizer modifications naturally lead to similar vocabularies and have smaller differences. Thus, there is no clearly best modification, as we discuss in Section 6.

	Priority Criterion	Merge Skip	1	2	3	4	5	6	7	8	9	10
1	$\mathcal{F}$ (BPE)	— (BPE)	0	8304	9168	9386	5848	6362	9580	10004	10088	10228
2	$\mathcal{F}$	$\mathcal{L} < 4$	8304	0	5572	5392	6132	5878	3292	6434	6472	6392
3	$\mathcal{F} \cdot \mathcal{L}$	—	9168	5572	0	372	6366	6316	5006	2666	2826	2894
4	$\mathcal{F} \cdot \mathcal{L}$	$\mathcal{L} = 1$	9386	5392	372	0	6562	6346	4842	2614	2764	2678
5	$\mathcal{F} \cdot \log(\mathcal{R}+1)$	—	5848	6132	6366	6562	0	576	4834	5892	5956	6086
6	$\mathcal{F} \cdot \log \mathcal{R}$	—	6362	5878	6316	6346	576	0	4454	5590	5618	5728
7	$\mathcal{F} \cdot \log(\mathcal{R}+1)$	$\mathcal{L} < 4$	9580	3292	5006	4842	4834	4454	0	4368	4352	4282
8	$\mathcal{F} \cdot \log(\mathcal{R}+1) \cdot \mathcal{L}$	—	10004	6434	2666	2614	5892	5590	4368	0	216	332
9	$\mathcal{F} \cdot \log \mathcal{R} \cdot \mathcal{L}$	—	10088	6472	2826	2764	5956	5618	4352	216	0	200
10	$\mathcal{F} \cdot \log \mathcal{R} \cdot \mathcal{L}$	$\mathcal{L} = 1$	10228	6392	2894	2678	6086	5728	4282	332	200	0

Table 14: Counts of different tokens in the trained tokenizers. The columns with numbers represent the models in the same order, enumerated by identifiers in the first column.

Priority Criterion	Merge Skip	Under-trained token examples
$\mathcal{F}$ (BPE)	— (BPE)	`0123351271903`, `_AtaSmartAttributeDisplayType`, `minCuSize`, `DISKinematics`
$\mathcal{F}$	Skip $\mathcal{L} < 4$	`IMARY`, `ientation`, `vanced`, `pendencies`, `ventory`
$\mathcal{F} \cdot \mathcal{L}$	—	`OfSurgeServices`, `AceHighFlush`, `PeriodoDeclara`, `OXWSMTGS`
$\mathcal{F} \cdot \mathcal{L}$	Skip $\mathcal{L} = 1$	`arency`, `ICENSE`, `ployment`, `ereum`, `avascript`
$\mathcal{F} \cdot \log(\mathcal{R}+1)$	—	`gridBagConstraintsPeriodoDeclara`, `HighFlush`, `ablytyped`
$\mathcal{F} \cdot \log \mathcal{R}$	—	`IntoConstraints`, `scalably`, `ilibj`, `VisualStyleBackColor`, `arency`
$\mathcal{F} \cdot \log(\mathcal{R}+1)$	Skip $\mathcal{L} < 4$	`AWSCloud`, `crets`, `ooser`, `NECTION`, `ereum`
$\mathcal{F} \cdot \log(\mathcal{R}+1) \cdot \mathcal{L}$	—	`_HandRankName`, `chordie`, `TINGS`, `ramimages`, `ekUSDI`
$\mathcal{F} \cdot \log \mathcal{R} \cdot \mathcal{L}$	—	`scriptors`, `TEGER`, `versation`, `FTWARE`, `clare`
$\mathcal{F} \cdot \log \mathcal{R} \cdot \mathcal{L}$	Skip $\mathcal{L} = 1$	`FTWARE`, `NECTION`, `IMARY`, `trieve`, `elcome`

Table 15: Examples of verified under-trained tokens in multilingual models.

H Under-Trained Tokens

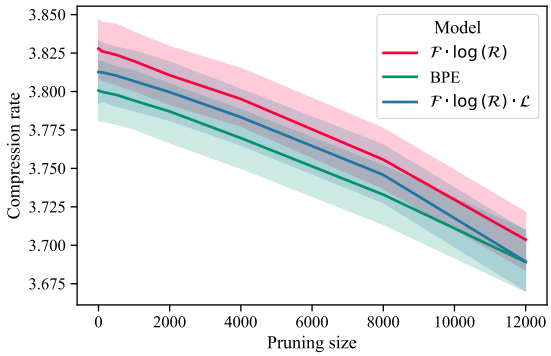
In Table 15, we show representative examples of under-trained tokens in the multilingual models. The criteria that allow single-language or single-repository still contain variable names among the under-trained tokens. The other criteria are better regularized and contain predominantly intermediate tokens among the under-trained tokens. We show in Section 5 that the number of such tokens is low when the general overfitting is reduced.

I Pruning and Compression

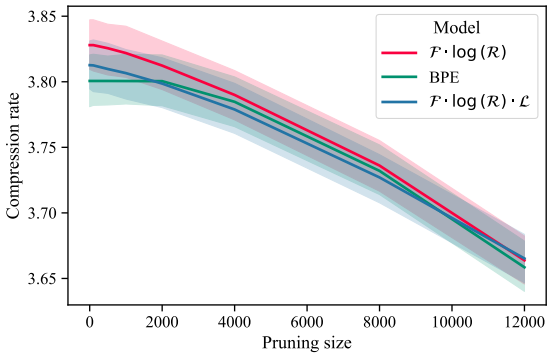
In Figures 9–26, we show the compression rate of pruned tokenizers per language in the order of subsample size in the training set using pruning starting from the end of the merge list, i.e., in the reversed merge order, and leaf-based pruning that uses the distance from $(0, 0)$ in under-trained token indicator space as a priority criterion. In the first case, for most languages, the compression rate of our regularized tokenizers is consistently better than or comparable to that of BPE. In the second case, for all languages, the first several thousand removed tokens have no effect on the compression rate, as they are over-fitted to the training sample and are not utilized. In lower-resource languages, the confidence intervals for compression rates are becoming closer together. However, for some languages, such as Lua, the model with the language component shows better compression, which may suggest that this provides better support for lower-resource languages. This corresponds well to our findings in Appendix G.

J LLM Usage Statement

In this work, we utilized AI tools for paraphrasing and grammar correction (Grammarly) and assistance with data visualization design (Gemini, ChatGPT). We also used Claude Code to prettify and document the paper code in a GitHub repository.

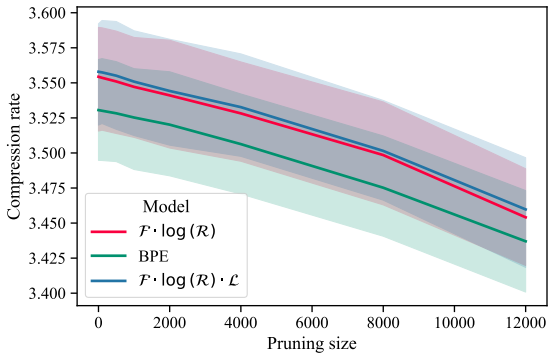


(a) Pruning order: reverse merge order.

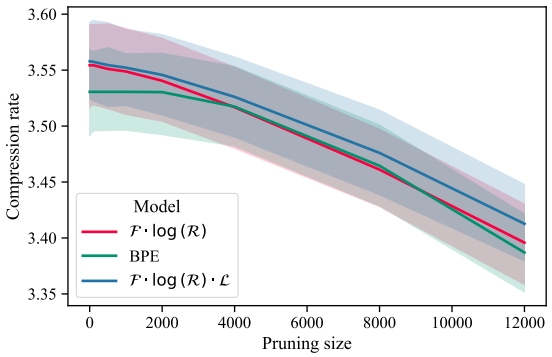


(b) Pruning order: under-trained first.

Figure 9: Compression rate for Java for tokenizers with applied pruning **(a)** in the reverse order of token ids and **(b)** starting from the tokens with the lowest under-trained indicator values (distance from (0, 0) in indicator space).

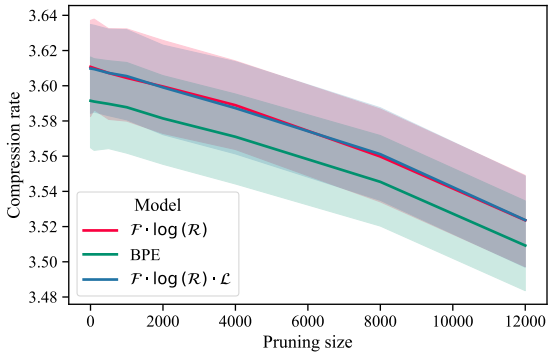


(a) Pruning order: reverse merge order.

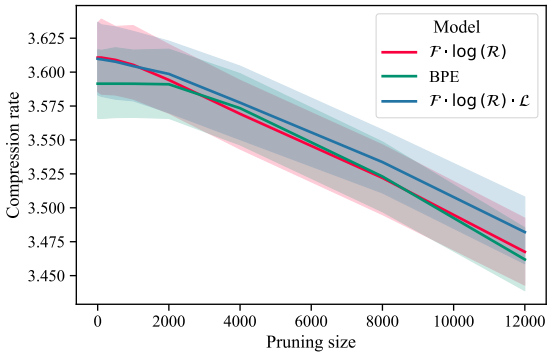


(b) Pruning order: under-trained first.

Figure 10: Compression rate for JavaScript for tokenizers with applied pruning **(a)** in the reverse order of token ids and **(b)** starting from the tokens with the lowest under-trained indicator values (distance from (0, 0) in indicator space).

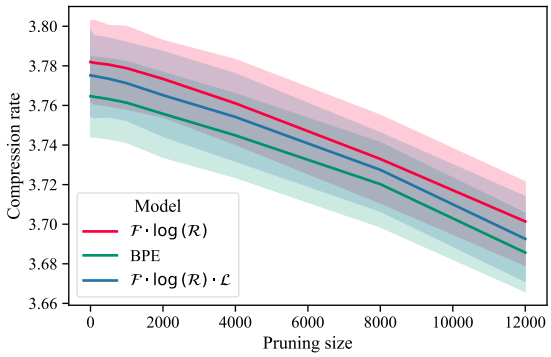


(a) Pruning order: reverse merge order.

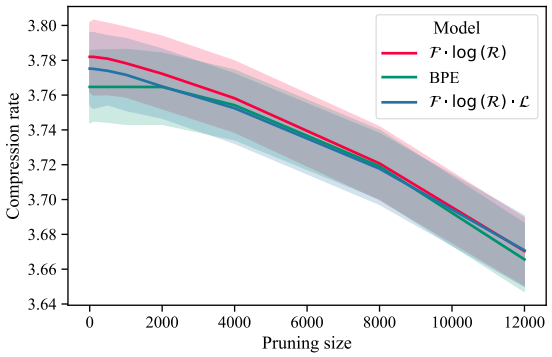


(b) Pruning order: under-trained first.

Figure 11: Compression rate for Python for tokenizers with applied pruning **(a)** in the reverse order of token ids and **(b)** starting from the tokens with the lowest under-trained indicator values (distance from (0, 0) in indicator space).

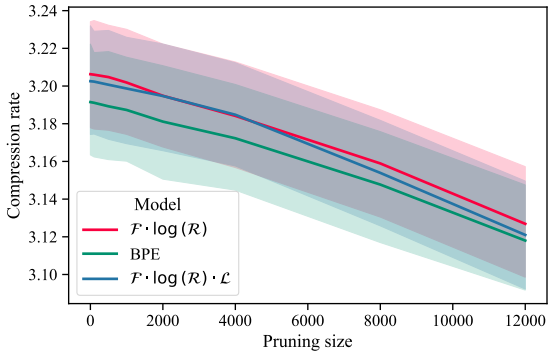


(a) Pruning order: reverse merge order.

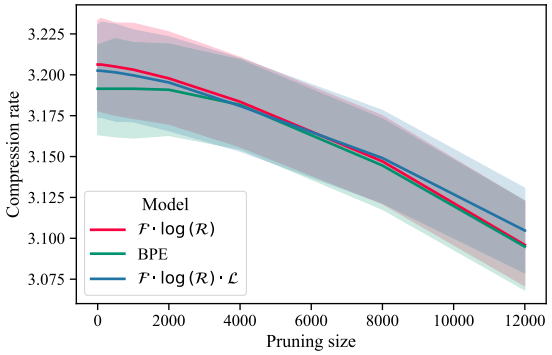


(b) Pruning order: under-trained first.

Figure 12: Compression rate for PHP for tokenizers with applied pruning **(a)** in the reverse order of token ids and **(b)** starting from the tokens with the lowest under-trained indicator values (distance from (0, 0) in indicator space).

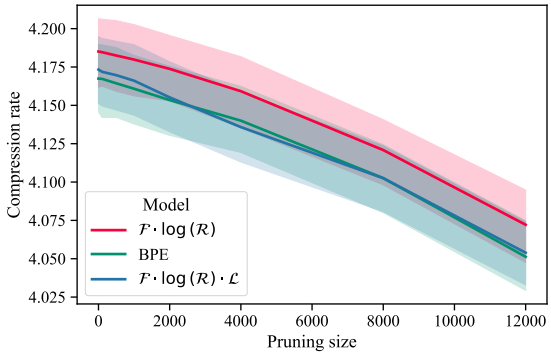


(a) Pruning order: reverse merge order.

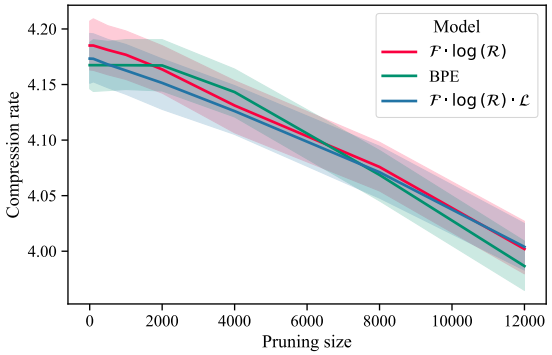


(b) Pruning order: under-trained first.

Figure 13: Compression rate for C++ for tokenizers with applied pruning **(a)** in the reverse order of token ids and **(b)** starting from the tokens with the lowest under-trained indicator values (distance from (0, 0) in indicator space).

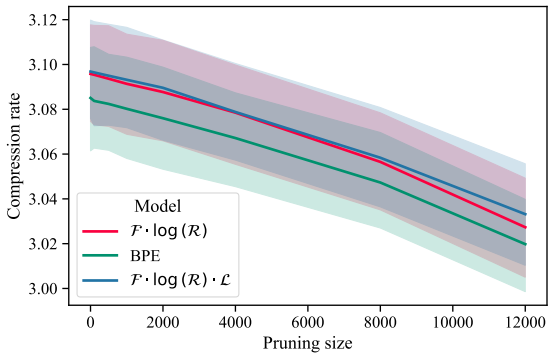


(a) Pruning order: reverse merge order.

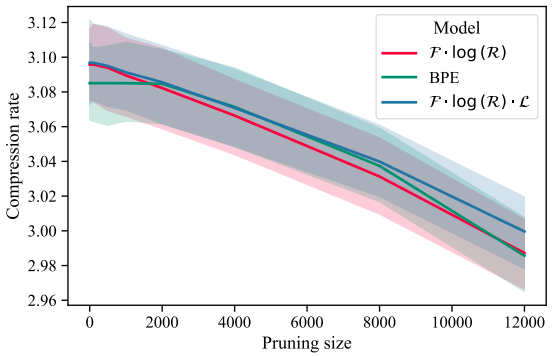


(b) Pruning order: under-trained first.

Figure 14: Compression rate for C# for tokenizers with applied pruning **(a)** in the reverse order of token ids and **(b)** starting from the tokens with the lowest under-trained indicator values (distance from (0, 0) in indicator space).

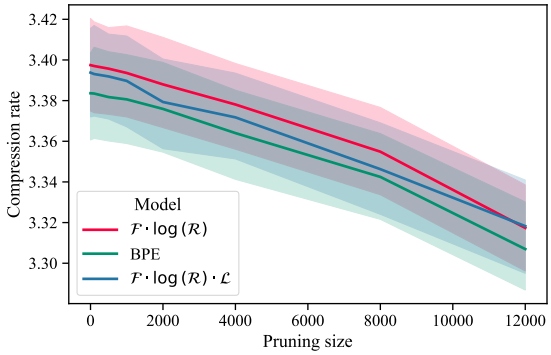


(a) Pruning order: reverse merge order.

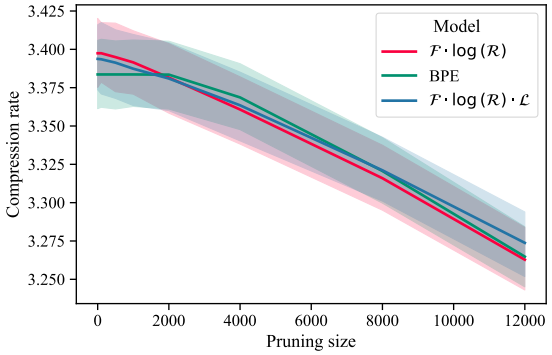


(b) Pruning order: under-trained first.

Figure 15: Compression rate for Go for tokenizers with applied pruning **(a)** in the reverse order of token ids and **(b)** starting from the tokens with the lowest under-trained indicator values (distance from (0, 0) in indicator space).

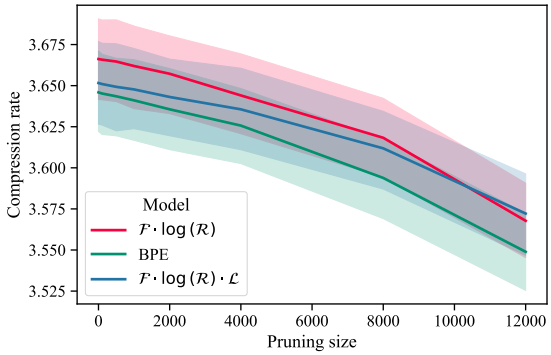


(a) Pruning order: reverse merge order.

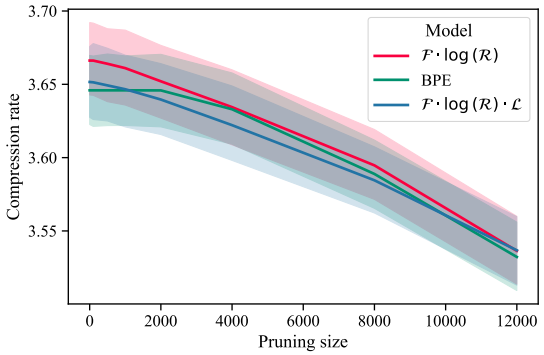


(b) Pruning order: under-trained first.

Figure 16: Compression rate for Rust for tokenizers with applied pruning **(a)** in the reverse order of token ids and **(b)** starting from the tokens with the lowest under-trained indicator values (distance from (0, 0) in indicator space).

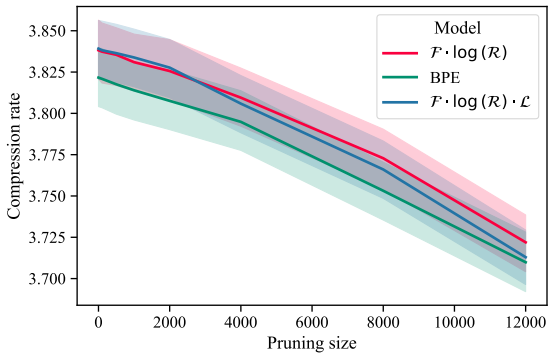


(a) Pruning order: reverse merge order.

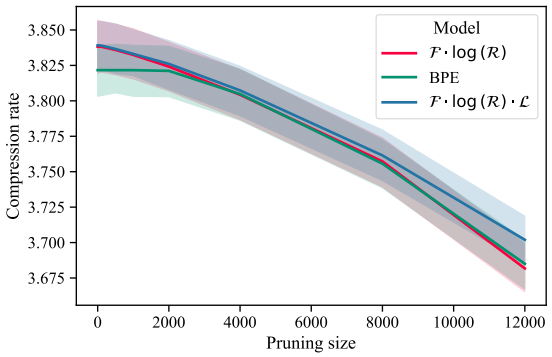


(b) Pruning order: under-trained first.

Figure 17: Compression rate for Ruby for tokenizers with applied pruning **(a)** in the reverse order of token ids and **(b)** starting from the tokens with the lowest under-trained indicator values (distance from (0, 0) in indicator space).

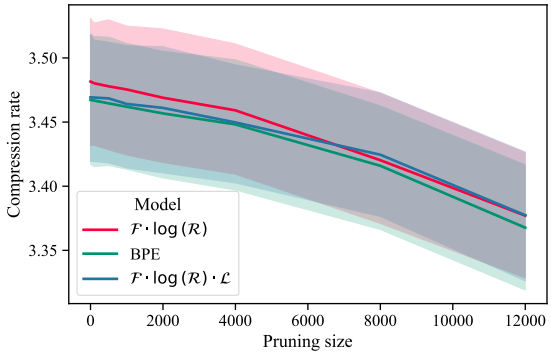


(a) Pruning order: reverse merge order.

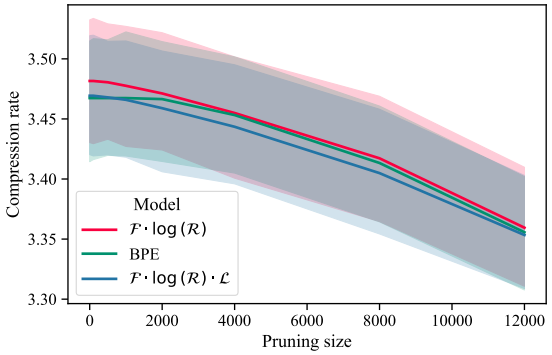


(b) Pruning order: under-trained first.

Figure 18: Compression rate for Kotlin for tokenizers with applied pruning **(a)** in the reverse order of token ids and **(b)** starting from the tokens with the lowest under-trained indicator values (distance from (0, 0) in indicator space).

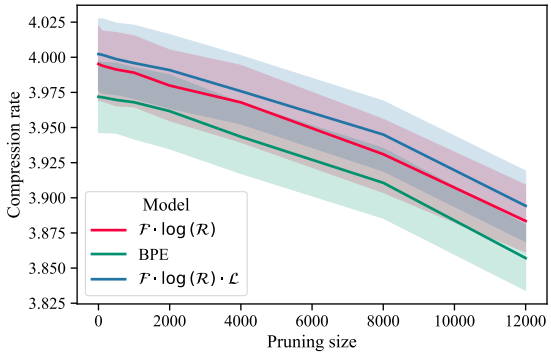


(a) Pruning order: reverse merge order.

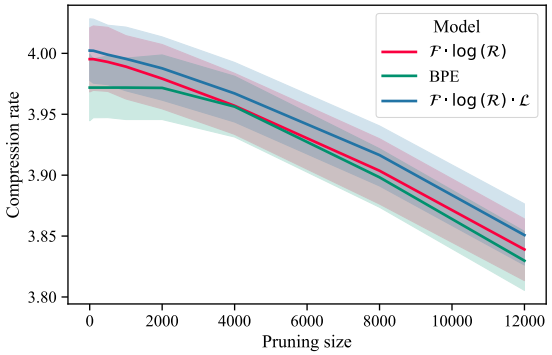


(b) Pruning order: under-trained first.

Figure 19: Compression rate for Scala for tokenizers with applied pruning **(a)** in the reverse order of token ids and **(b)** starting from the tokens with the lowest under-trained indicator values (distance from (0, 0) in indicator space).

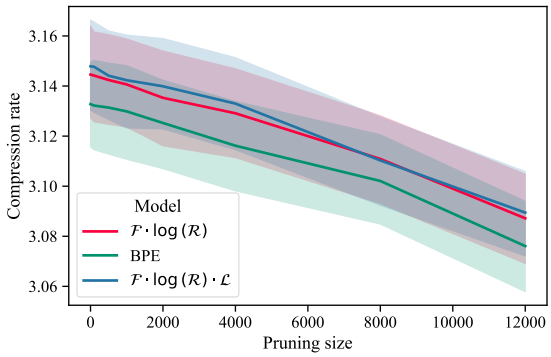


(a) Pruning order: reverse merge order.

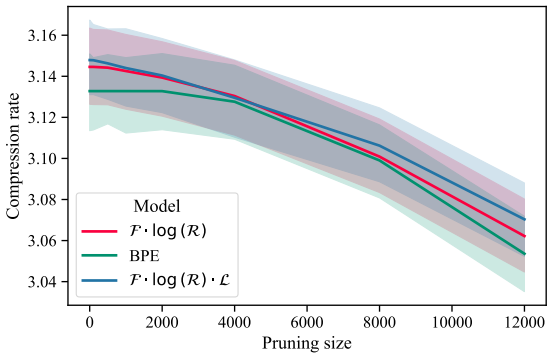


(b) Pruning order: under-trained first.

Figure 20: Compression rate for Swift for tokenizers with applied pruning **(a)** in the reverse order of token ids and **(b)** starting from the tokens with the lowest under-trained indicator values (distance from (0, 0) in indicator space).

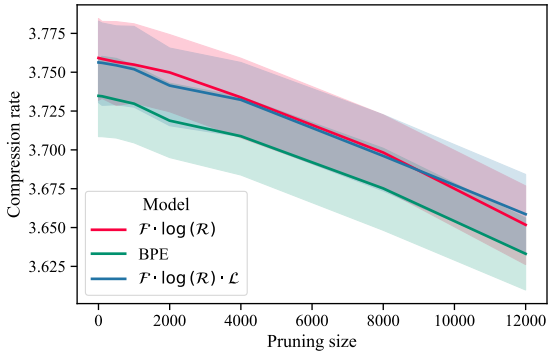


(a) Pruning order: reverse merge order.

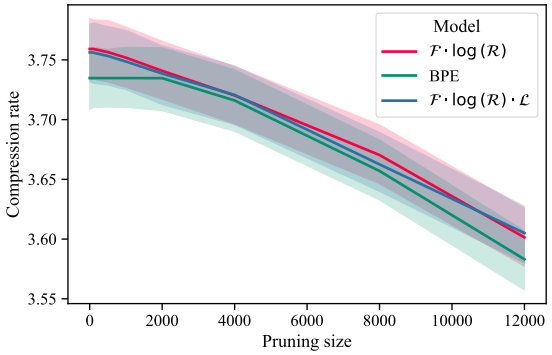


(b) Pruning order: under-trained first.

Figure 21: Compression rate for Vue for tokenizers with applied pruning **(a)** in the reverse order of token ids and **(b)** starting from the tokens with the lowest under-trained indicator values (distance from (0, 0) in indicator space).

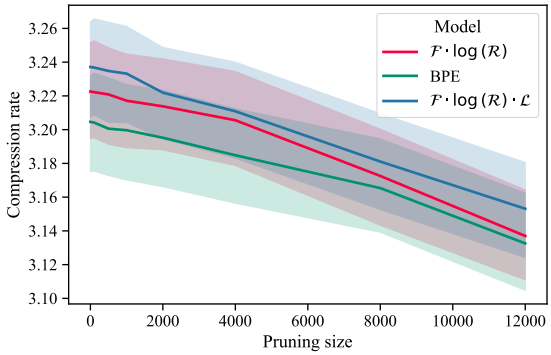


(a) Pruning order: reverse merge order.

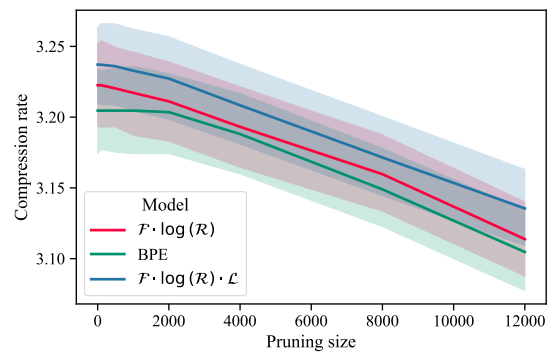


(b) Pruning order: under-trained first.

Figure 22: Compression rate for Dart for tokenizers with applied pruning **(a)** in the reverse order of token ids and **(b)** starting from the tokens with the lowest under-trained indicator values (distance from (0, 0) in indicator space).

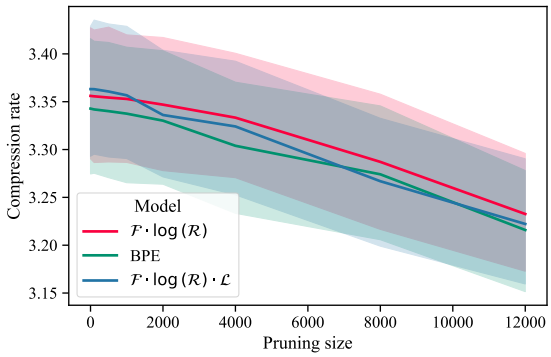


(a) Pruning order: reverse merge order.

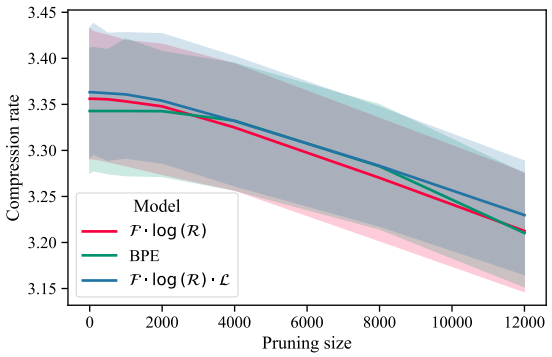


(b) Pruning order: under-trained first.

Figure 23: Compression rate for Lua for tokenizers with applied pruning **(a)** in the reverse order of token ids and **(b)** starting from the tokens with the lowest under-trained indicator values (distance from (0, 0) in indicator space).

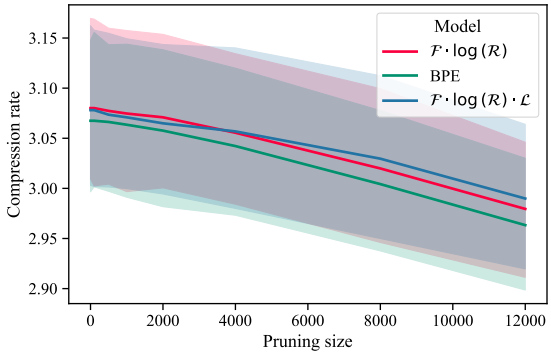


(a) Pruning order: reverse merge order.

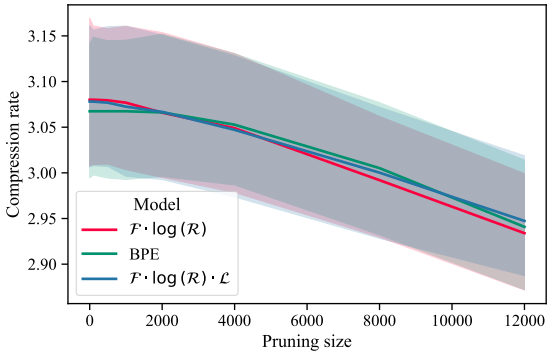


(b) Pruning order: under-trained first.

Figure 24: Compression rate for Haskell for tokenizers with applied pruning **(a)** in the reverse order of token ids and **(b)** starting from the tokens with the lowest under-trained indicator values (distance from (0, 0) in indicator space).

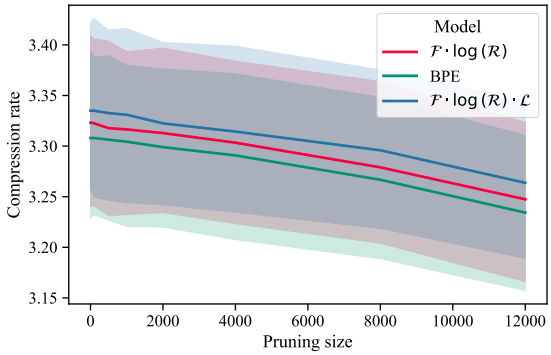


(a) Pruning order: reverse merge order.

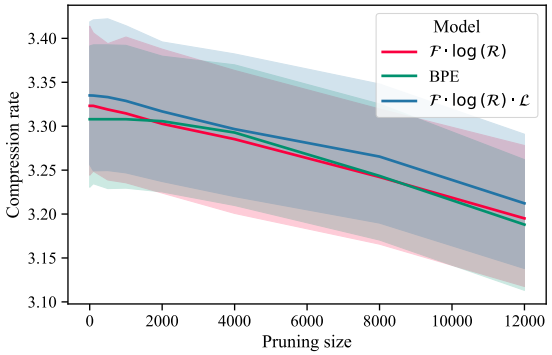


(b) Pruning order: under-trained first.

Figure 25: Compression rate for Julia for tokenizers with applied pruning **(a)** in the reverse order of token ids and **(b)** starting from the tokens with the lowest under-trained indicator values (distance from (0, 0) in indicator space).



(a) Pruning order: reverse merge order.



(b) Pruning order: under-trained first.

Figure 26: Compression rate for OCaml for tokenizers with applied pruning **(a)** in the reverse order of token ids and **(b)** starting from the tokens with the lowest under-trained indicator values (distance from (0, 0) in indicator space).