Getting the most out of your tokenizer for pre-training and domain adaptation

Gautier Dagan [1] **Gabriel Synnaeve** [2] **Baptiste Rozière** [2]

Abstract

Tokenization is an understudied and often neglected component of modern LLMs. Most published works use a single tokenizer for all experiments, often borrowed from another model, without performing ablations or analysis to optimize tokenization. Moreover, the tokenizer is generally kept unchanged when fine-tuning a base model. In this paper, we show that the size, pre-tokenization regular expression, and training data of a tokenizer can significantly impact the model's generation speed, effective context size, memory usage, and downstream performance. We train specialized Byte-Pair Encoding code tokenizers, and conduct extensive ablations on the impact of tokenizer design on the performance of LLMs for code generation tasks such as HumanEval and MBPP, and provide recommendations for tokenizer hyper-parameters selection and switching the tokenizer in a pre-trained LLM. We perform our experiments on models trained from scratch and from pre-trained models, verifying their applicability to a wide range of use-cases. We find that when fine-tuning on more than 50 billion tokens, we can specialize the tokenizer of a pre-trained LLM to obtain large gains in generation speed and effective context size.

1. Introduction

Tokenizers transform raw sequences of text into tokens, and are an essential part of most modern language models. They are generally built using the Byte-Pair Encoding (BPE) algorithm (Sennrich et al., 2016), and more rarely with the Unigram algorithm (Kudo, 2018). Building a tokenizer is one of the first steps of any language modeling project. Modifying the tokenizer impacts everything downstream, and is generally more cumbersome than performing ablations on other hyper-parameters or modeling features. Hence, practitioners often report results based on a single set of tokenizer hyper-parameters.

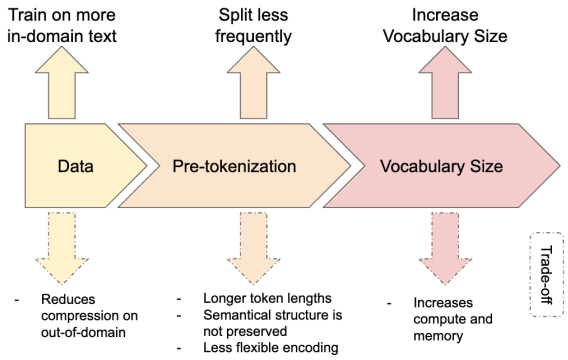

Figure 1. Three ways to increase in-domain compression in a BPE tokenizer with their respective trade-offs.

This is especially true for projects fine-tuning a pre-trained LLM for a specific task or domain. The tokenizer of the base model is generally unchanged, leading to tokenizers being applied to domains for which they are sub-optimal, hurting efficiency and potentially the performance of the final model. For instance, Code Llama (Rozière et al., 2023) re-uses the tokenizer that was used by its base model, Llama 2 (Touvron et al., 2023b), which uses the tokenizer from the original Llama model (Touvron et al., 2023a). This means that Code Llama, while being a popular model fine-tuned on a domain specific task, is still limited by the decisions taken during the pre-training of the original base model. However, higher-compression code tokenizers exist. For instance, the InCoder (Fried et al., 2023) tokenizer is trained on source code, has a larger vocabulary size, and uses less restrictive pre-processing rules. As a result, it uses 25% less tokens than the Llama tokenizer on average when encoding source code (see Table 1).

Efficiency in token usage can have a significant impact on LLM inference and training. In the context of modern LLMs, where the inference budget is non-negligible (Touvron et al., 2023a), enhancements in compression like these lead to more efficient inference in both FLOPS and memory usage. Compression also increases the effective context length of the model, defined as the number of characters the

[1]University of Edinburgh, Edinburgh, UK [2]Meta AI, Paris, France. Correspondence to: Gautier Dagan <gautier.dagan@ed.ac.uk>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

model can ingest on average. Conversely, high-compression tokenizers can negatively impact the performance of the model in other ways. While the compute cost of increasing the size of the vocabulary is negligible in practice for LLMs, it can significantly impact memory usage, especially for smaller models. Compute is also critical limiting factor when training LLMs, and high-compression tokenizers allow to train on more text for a fixed compute budget.

Existing tokenizer studies perform experiments at a much smaller scale than what is typical for modern LLMs (Gowda & May, 2020; Chirkova & Troshin, 2023), or focus on multilingual tasks (Rust et al., 2021; Limisiewicz et al., 2023; Zouhar et al., 2023). It is not clear whether these findings transfer to models with greater capacity and trained on more tokens. Similarly, some previous work has looked at adapting tokenizers in pre-trained LLMs (Mosin et al., 2023; Gee et al., 2022; 2023), but only on smaller LLMs ($<$ 1B parameters) and vocabulary sizes ($<$ 32k tokens). Previous work also did not continue pre-training with the masked language model objective, but instead fine-tuned models directly on a downstream task.

In this paper, we focus our experiments on modern code LLMs of 1.5B and 7B parameters. Our contributions are as follows:

- We compare popular code tokenizers, clarifying their respective performances and trade-offs made.
- We study the impact of vocabulary size, pre-tokenization regular expression on compression and downstream code generation performance when fine-tuning and training from scratch. We observe that the pre-tokenization can substantially impact both metrics and that vocabulary size has little impact on coding performance.
- For adapting existing models, we show that the tokenizer can be changed with little impact to downstream performance when training on 50B tokens or more.

In section 2, we detail three ways to increase tokenizer compression, and propose methods to calculate inference and memory optimal vocabulary sizes. In section 3, we study the impact of fine-tuning and training from scratch an LLM with a different tokenizer, and evaluate the effects of tokenizer settings on downstream code generation.

2. Compression trade-offs

We identify three main levers impacting the downstream compression of a tokenizer on a specific domain (see Figure 1). The first one is the data used to train the tokenizer, where using data sampled from the in-domain distribution will increase in-domain compression. The second lever is

		NSL (↓)			
	Size	Avg.	Code	Eng.	Mult.
GPT-2 (Radford et al., 2019)	50k	1.13	1.19	0.86	1.33
DeepSeek Coder (DeepSeek AI, 2023)	32k	1.06	1.00	0.98	1.19
Llama (Touvron et al., 2023a)	32k	1.00	1.00	1.00	1.00
CodeGen (Nijkamp et al., 2023)	50k	1.05	0.95	0.86	1.33
CodeT5 (Wang et al., 2021)	32k	1.29	0.94	1.11	1.83
SantaCoder (Allal et al., 2023)	49k	1.04	0.88	1.07	1.17
StarCoder (Li et al., 2023)	49k	0.99	0.87	1.04	1.07
Replit Code (Replit, 2023)	32k	1.00	0.85	1.06	1.10
GPT-4 (OpenAI, 2023)	100k	0.85	0.75	0.84	0.95
InCoder (Fried et al., 2023)	50k	1.03	0.74	1.02	1.31
Ours					
Punct	32k	0.98	0.86	0.96	1.11
Punct	64k	0.90	0.82	0.89	0.99
Punct	80k	0.88	0.81	0.88	0.95
Punct	100k	0.86	0.81	0.86	0.92
GPT-4	32k	0.97	0.81	0.97	1.13
GPT-4	64k	0.89	0.76	0.90	1.01
GPT-4	80k	0.87	0.75	0.88	0.98
GPT-4	100k	0.85	0.74	0.86	0.94
Identity	32k	0.92	0.69	0.89	1.16
Identity	64k	0.82	0.63	0.79	1.04
Identity	80k	0.80	0.61	0.76	1.01
Identity	100k	0.77	0.59	0.74	0.98
Merged	80k	0.90	0.80	0.95	0.94

Table 1. Comparison of popular code tokenizers by their Normalized Sequence Length (NSL), in this case NSL calculated against the Llama tokenizer, and indicates the average tokenized sequence length that a tokenizer would produce compared to Llama (see section 2.1). The lower the NSL, the more efficient the tokenizer is at compressing the dataset. For example, on our Code subset, the InCoder (Fried et al., 2023) tokenizer uses on average 26% less tokens than the Llama tokenizer.

the pre-tokenization scheme, which can be written as a regular expression defining how the text is split before it is passed to the BPE tokenizer. Splitting sequences prevents BPE from merging certain tokens, for instance splitting on white spaces means that a token cannot span two space-separated words. It leads to shorter tokens and thus worse compression rates, but is generally done to improve downstream performance. Finally, increasing the vocabulary size leads to higher compression at the cost of compute and memory.

It is important to note that higher compression rates could also lead to deteriorated downstream performance, since shorter sequences give less effective FLOPs to a model to reason (Goyal et al., 2023). This is a consequence of the modern Transformer decoder architecture in which *every token requires an additional forward pass to generate*. Therefore even seemingly low-information tokens might still provide gains on downstream task. This is evidenced by Goyal et al. (2023), who propose Pause Tokens, special empty tokens added to the context to enable the model to 'pause' its reasoning and add FLOPs during inference.

2.1. Compression metrics

Compression is always measured as a ratio of a quantity with respect to another. We measure two compression metrics, the first, Normalized Sequence Length (NSL), compares the compression of a given tokenizer with respect to our baseline Llama tokenizer. The second, is the average number of Bytes per Token, and is calculated by dividing the number of UTF-8 bytes by the number of tokens produced by the tokenizer on a given text.

Formally, we define NSL $c_{\frac{\lambda}{\beta}}$ as the ratio between the length of an encoded sequence from a tokenizer T_λ and a tokenizer T_β. For N examples taken from a dataset D:

$$c_{\frac{\lambda}{\beta}} = \frac{\sum_{i=1}^{N} \text{length}(T_\lambda(D_i))}{\sum_{i=1}^{N} \text{length}(T_\beta(D_i))}$$

We use the Llama tokenizer (Touvron et al., 2023a) as our reference T_β. In other words, if T_λ has an NSL of 0.75 on average, it means that sequences encoded with T_λ contain 25% less tokens on average than those encoded with Llama.

We use the public datasets CCnet (Wenzek et al., 2020), Wikipedia, and the Stack (Kocetkov et al., 2022) for tokenizer training and evaluation. We divide data into three categories: English, code, and multilingual data (see Appendix A). Multilingual data consists of non-English texts from Wikipedia and includes 28 languages. The code category includes 30 different programming languages. For each subset (programming language, natural language) of each dataset, we hold-out 1000 examples (files) and use these to compute the compression ratio. The compression ratio of a category is calculated as an average over all its respective subsets. This ensures that subsets which contains longer sequences, for instance C++ code, is weighted equally against subsets that contain shorter sequences in average. The overall average NSL is calculated as the average over the held-out code, English and multilingual data.

2.2. Algorithm

We use the BPE (Sennrich et al., 2016) tokenization algorithm as it is the most commonly used to train general and code-specific tokenizers. We considered two popular libraries implementing BPE training, Google's Sentencepiece (Kudo & Richardson, 2018) and HuggingFace tokenizers (Wolf et al., 2019). Since we measure the effects of different pre-tokenization schemes on code, we opt to use HuggingFace's tokenizer library as it supports a regular expression-based pre-tokenization and better handles special formatting characters such as tabs and new lines.

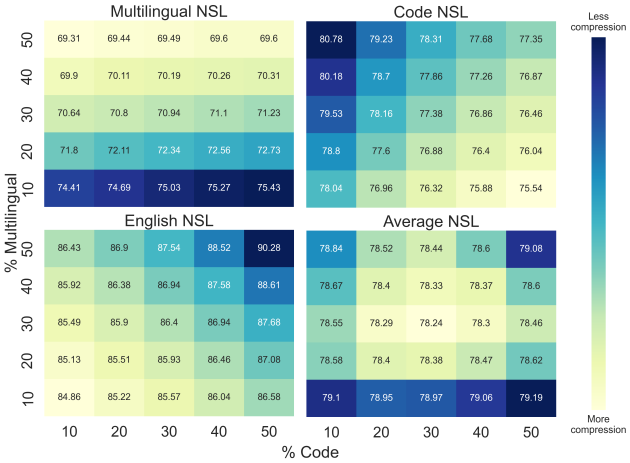

Figure 2. Tokenizers trained with different % of code, English, multilingual data. Unsurprisingly, training on code improves code compression, training on multilingual data improves multilingual compression, and training on an even mix of all three subset leads to the best average compression.

2.3. Data

Perhaps unsurprisingly, the data used to train a BPE tokenizer significantly impact its compression on evaluation datasets. We train tokenizers on different dataset mixes and compare the compression (NSL) obtained on a held-out sets. We fix the number of characters used to train learn the BPE tokenizer to 10 Billion, and vary only the percentage of code and multilingual training data in the training dataset. We keep all the other hyper-parameters constant.

Figure 2 shows the NSL of our trained tokenizers on three held-out sets for Multilingual, Code and English. We measure the average NSL on all subset, as well as the average over the three (Average NSL). As expected, we find that training on more code improves code compression, training on multilingual data improves multilingual compression, and training on an even mix of all three subset leads to the best global average compression. Figure 2 reinforces the notion that *tokenizers should be trained on the data mix that they are expected to see during training/inference*. We also observe that the NSL on any given subset only varies by 5 to 6 percentage points. For example when 50% of the data is multilingual, the NSL is only improved by about 5% compared to when 10% of the data is multilingual.

For the rest of this paper, since our target domain is Code Generation, we train all tokenizers (shown in Table 1) on a data distribution of 70% code and 30% English.

2.4. Pre-tokenization

BPE (Sennrich et al., 2016) operates by iteratively merging frequent adjacent characters or character sequences to build

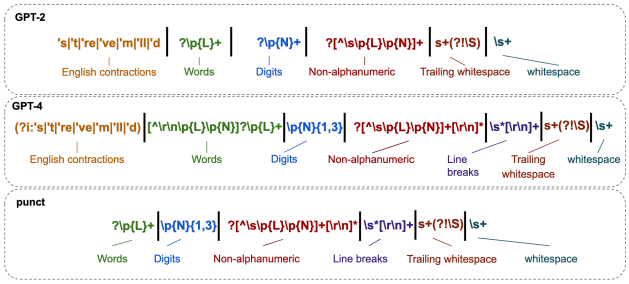

Figure 3. The GPT-2 (Radford et al., 2019) and GPT-4 (OpenAI, 2023) pre-tokenization regular expressions decomposed into functional sub-parts, and another version dubbed Punct which we introduce to ablate some of the changes introduced in GPT-4. Punct does away with the English-specific contractions and prevents certain whitespace and punctuation tokens such as `\t` or `.` to be encoded at the start of an alpha-only token (see Appendix G for an example).

a vocabulary from sub-word units. When applied naively, this method leads to tokens forming around common phrases or sentences, which could be sub-optimal for certain tasks. *Pre-tokenization* is a pre-processing step that happens before passing the text to the tokenization algorithm. Most commonly, this step involves breaking down text into more granular chunks. This can be based on linguistic rules, such as splitting on punctuation marks or spaces, to ensure that the individual learned tokens are meaningful and promote compositional re-use.

For instance, consider that without any pre-tokenization, entire phrases or common occurrences such as dates (`2022`) can be represented as a single token. While this might be optimal in terms of compression, this introduces complexities for the downstream LLM. For instance, if tasked with an arithmetic problem, arbitrary tokens such as `2022` forces the model to learn arithmetic for every token independently. In comparison, a tokenizer splitting on individual digits would encode `2022` as `2`, `0`, `2`, `2` separately, and could therefore learn to generalize basic arithmetic. Previous works have also shown that digit tokenization can significantly impact arithmetic performance (Nogueira et al., 2021; Thawani et al., 2021).

Note that to keep our tokenization scheme perfectly reversible, we abstain from any form of normalization in our pre-tokenization step (see Appendix D).

Pre-tokenizers based on regular expressions. Regular expressions provide a powerful mechanism for defining patterns for text segmentation. They are often used to create custom pre-tokenizers tailored to specific data or tasks. GPT-2 (Radford et al., 2019), in particular, introduced a large regular expression (shown in Figure 3) to split the text into chunks before applying BPE. In GPT-4 (OpenAI, 2023), that regular expression was expanded to capture more specific groups, for instance limiting the number of digits allowed in a token to three instead of an unbounded number. Figure 3) presents a detailed breakdown of the regular expression used in GPT-2, GPT-4, and a simplified regular expression we refer to as Punct (see Appendix E). We use Punct to test whether a stronger separation between syntax and semantics could simplify the language generation task and ultimately translate to greater downstream performance.

2.5. Vocabulary Size

Vocabulary size, or number of tokens, is a key hyper-parameter that impacts the cost-efficiency of a Transformer model. While a larger vocabulary increases the cost per decoding step, it reduces both the memory needed for attention cache and the computation for generating a sentence. This increase in cost primarily affects the embedding and output layers. Therefore, in larger LLMs, the relative impact of a larger vocabulary on the overall parameter count becomes negligible (see Appendix B). Consequently, for sufficiently large models, the benefits of a larger vocabulary, in terms of reduced total compute and memory requirements at inference, can outweigh the costs.

Large vocabulary sizes could also have adverse effects on downstream performance: with a large vocabulary every token is seen less on average by the model. This is a natural consequence of Zipf's law (Zipf, 1949). Therefore, we test whether tokenizer vocabulary size impacts performance in Section 3.2.

2.5.1. Optimal Vocabulary Size

Figure 4 (top left) shows the Code NSL curve for a tokenizers trained on the same set of data, with varying vocabulary sizes from 10k to 256k. We normalize the plot to a vocabulary size of $32,000$ (Code NSL@32k). The gains in compression exponentially decrease as the vocabulary size increases, which indicates that there is an optimum point for a given downstream application where the additional token is not worth its added cost in compute or memory.

Inference Optimal We run experiments with different model sizes to measure to the effects of the vocabulary size on inference time with a fixed sequence length. In Figure 4 (top middle), we show these observations and plot the linear regressions found for each LLM size. We normalize the observations and predictions to a vocabulary size of 32k. Since the NSL describes the length of a sequence, it directly affects inference time. We thus calculate the trade-off cost as the product between NSL@32k (top left) and the normalized inference time for each vocabulary size (top middle). We plot these trade-off curves and find the minimum point for each LLM size in Figure 4 (top right). We find optimal inference time vocabulary size to grow with the size of the

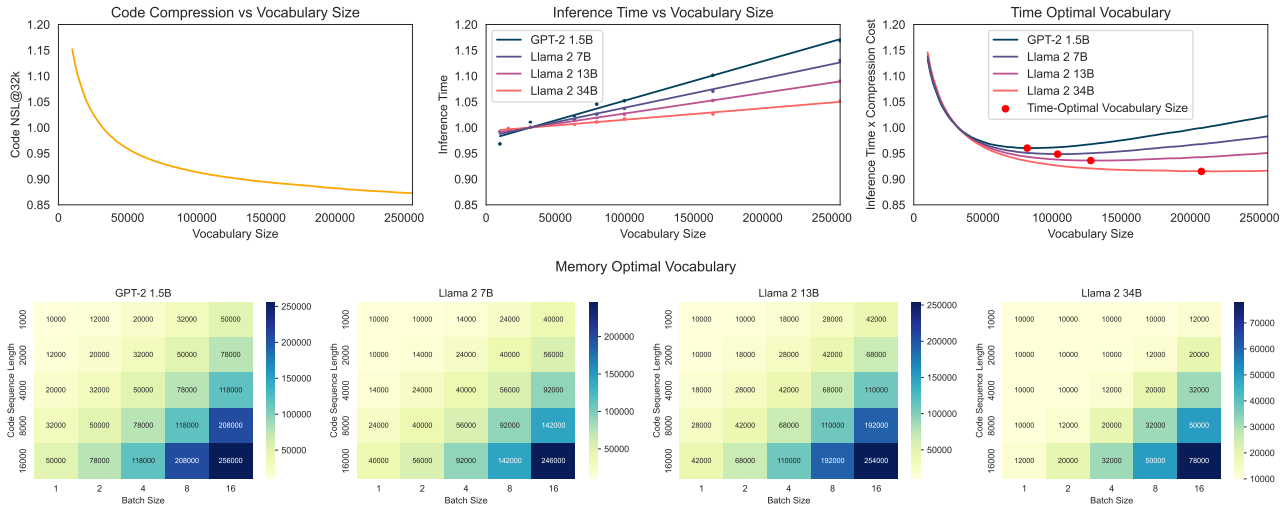

Figure 4. **(top left)** For given fixed set of tokenizer settings, we measure the Code NSL of different vocabulary sizes. We set the reference point to the tokenizer trained @32k tokens to compare against. **(top middle)** We measure the inference time for a set of vocabulary sizes and models with a fixed sequence length of 4096, and plot a linear regression over observations. We normalize predictions to a vocab of 32k. **(top right)** By combining the compression and inference time trade-offs, we obtain a simple cost function that describes an optimal inference time. **(bottom)** We use equation 4 to find the memory optimal vocabulary size for different models. Llama 2 34B uses grouped-query attention, which significantly reduces the cache's memory usage and the memory-optimal vocabulary size.

LLM. For LLMs like Llama 30B, we prefer even small gains in tokenizer compression, despite additional tokens in the final softmax, because of the high base cost of the forward pass. Note that these calculations are heavily dependent on both hardware and software optimizations.

Memory Optimal The memory costs of running a LLM at inference time are mostly due to the weights of the model itself and its attention cache. At inference time, if the input and output matrices are not shared, the impact of increasing the size of the vocabulary v on the number of additional parameters is:

$$M(v) = 2 \times dim \times v \tag{1}$$

Recall that sequence size is affected by the compression of the tokenizer, which is itself affected by the vocabulary size. We can use the NSL@32k metric calculate the compressed length of a code sequence s given a vocabulary size v and the length of a sequence l_{32k} encoded at 32k:

$$s(v) = l_{32k} \times NSL@32k \tag{2}$$

Therefore the effects of vocabulary size v on the number of parameters to keep in the cache depends on the batch size b, number of layers n, the hidden dimension dim, the number of kv heads and the sequence length $s(v)$:

$$C(v) = 2 \times n \times b \times dim \times \frac{n_kv_heads}{n_heads} \times s(v) \tag{3}$$

Llama 2 34B and 70B use grouped-query attention (Ainslie et al., 2023), with only 8 kv heads, which significantly reduce the number of parameters kept in the cache compared to the 7B and 13B versions. We also assume model and cache parameters to be stored at the same precision level. We calculate the total effect of v on memory T as:

$$T(v) = M(v) + C(v) \tag{4}$$

Figure 4 (bottom) shows the memory optimal vocabulary sizes for models under different sequence lengths and batch sizes. For short sequences ($l_{32k} = 1000$) and small batches ($b = 1$) the gains in compression from expanding the vocabulary is not worth the additional memory costs it incurs. However, when using longer sequences or larger batches, the memory savings from the cache $C(v)$ are worth the additional memory costs $M(v)$ of an expanding vocabulary.

3. Code Tokenizers Experiments

We ask the question: *what would have happened if Code Llama had switched its tokenizer during fine-tuning*? To answer this question, we first train two GPT-2 XL 1.5B base models (Radford et al., 2019) with the Llama tokenizer which we refer to as NL 1.5B and Code 1.5B respectively. NL 1.5B is trained for 1T tokens on a general data-mix resembling that of Llama 2 and Code 1.5B is further fine-tuned from NL 1.5B for 500B tokens using a code-specific data-mix similar to that of Code Llama (see Appendix H for more details).

regexp^{Init}	Token-Equivalent (500B)						Word-Equivalent					
	HumanEval			MBPP			HumanEval			MBPP		
	Pass@1	Pass@100	Compile@1	Pass@1	Pass@100	Compile@1	Pass@1	Pass@100	Compile@1	Pass@1	Pass@100	Compile@1
Identity^{R}	15.8%	49.9%	96.5%	21.7%	66.7%	99.0%	14.9%	47.5%	96.3%	19.4%	64.9%	98.7%
Punct^{R}	18.2%	58.7%	97.4%	24.6%	71.2%	99.5%	17.6%	55.5%	96.9%	24.4%	71.0%	99.4%
GPT-4^{R}	17.8%	54.6%	99.0%	25.2%	70.7%	99.1%	17.1%	53.7%	97.2%	23.9%	70.0%	99.1%
Identity^{NL}	18.0%	60.1%	97.5%	24.5%	67.2%	99.1%	17.1%	56.1%	97.5%	22.7%	67.2%	98.2%
Llama^{NL}	20.5%	66.1%	98.4%	28.0%	71.2%	99.4%	20.5%	66.1%	98.4%	28.0%	71.2%	99.4%
Punct^{NL}	21.1%	63.5%	97.6%	28.6%	72.6%	99.5%	20.7%	62.1%	97.1%	28.2%	72.0%	99.4%
GPT-4^{NL}	20.5%	65.3%	98.0%	27.2%	70.8%	99.4%	19.7%	63.7%	97.8%	26.8%	69.1%	99.4%
Identity^{C}	18.8%	66.1%	96.7%	23.4%	67.8%	99.4%	16.5%	61.8%	95.4%	22.3%	66.9%	99.2%
Llama^{C}	22.8%	68.6%	98.8%	30.1%	74.4%	99.4%	22.8%	68.6%	98.8%	30.1%	74.4%	99.4%
Punct^{C}	22.2%	68.7%	98.1%	29.8%	73.6%	99.4%	22.3%	67.3%	97.3%	28.9%	71.8%	99.4%
GPT-4^{C}	21.2%	67.5%	97.7%	28.9%	74.9%	99.4%	21.4%	67.3%	97.8%	28.5%	74.2%	99.4%

Table 2. We report the performance of fine-tuned 1.5B models using different tokenizers and base models on our two task generation tasks (HumanEval and MBPP) after 500B tokens seen and at word-equivalent levels. Word-equivalent compares the models after being trained on the same amount of information as measured by the number of characters. Due to added compression of our tokenizers compared to Llama, the word-equivalent evaluation corresponds to training on less tokens and using less compute than the token-equivalent evaluation. All tokenizers presented here are of vocabulary size 32k – see Table 1 for compression statistics for each. ^{R} indicates a random initialization, ^{NL} indicates that the base model used is our NL 1.5B model, and ^{C} indicates that the base model is Code 1.5B.

We use the HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) datasets to test the downstream effect of tokenizer change on LLMs. For all evaluations, we generate $n = 200$ samples per example with a temperature of 0.6 and $top_p = 0.95$, and use the unbiased estimator detailed in Chen et al. (2021) to calculate Pass@1, Pass@100, and Compile@1. The Pass@k score (Kulal et al., 2019; Rozière et al., 2020; Chen et al., 2021) measures the semantic correctness of a code snippet, by checking whether at least one of the k generated solution passes **all** the available unit tests. Similarly, the Compile@k metric measures whether at least one of the k generated solutions compiles. Note that we also use *token healing* for all generations, which is a simple decoding trick to align the prompt along token boundaries (see Appendix K.1).

We report fine-tuning NL 1.5B and Code 1.5B, changing the tokenizer to our tokenizers of vocabulary size 32k (see Table 1 for compression statistics) or keeping the Llama tokenizer constant. See Table 2 for the downstream performance on code generation of different base model/tokenizer configurations after fine-tuning. For all tokenizer fine-tuning, whenever applicable, we apply Fast Vocabulary Transfer (FVT) (Gee et al., 2022) to initialize the weights of the new embeddings.

The tokenizer compression impacts how much data the model sees for a given number of tokens (the amount of compute). As compute is often the primary constraint in training LLMs, we believe that assessing the *token-equivalent* downstream performance, measured after training with the same number of tokens (500B), offers a fairer comparison between models. However, we also report *word-equivalent* performance in which compares models trained on the same number of characters.

Table 2 shows that when training a LLM from scratch on the same dataset (^{R}), we obtain worse performance than if we use NL as the base model (^{NL}). This shows that the pre-trained model weights from the base model are still leveraged after a tokenizer change. We see that this is the case as well when starting from a Code Llama type model (^{C}), where we get the best performance on the end-task when using Code 1.5B as the base model regardless of tokenizer. Llama ^{NL} is analogous to the original Code Llama by Rozière et al. (2023) – a Llama base model trained for 500B tokens on code without changing the tokenizer. Llama ^{C} would be that same Code Llama model fine-tuned for 500B more code tokens, and indicates further fine-tuning can still provide gains in code-generation performance. Figure 5 also shows these dynamics, where the HumanEval Pass@1 is improved when moving from ^{R} to ^{NL} to ^{C}.

While the Identity tokenizer has the greatest compression out of the tokenizers evaluated, it results in clear deteriorated performance on downstream code generation tasks. Compared to the Llama tokenizer, the 32k Identity model compresses code 30% more efficiently but its downstream performance is significantly worse on all metrics. Moreover, forgoing token healing (see Appendix K.1) is especially detrimental with the Identity tokenizer.

In terms of differences between the other tokenizers, we see in Table 2 that, at a vocabulary size of 32k, both Punct ^{NL} and GPT-4 ^{NL} obtain similar performance as Llama ^{NL}. On some metrics, such as Pass@1, Punct ^{NL} even surpasses the Llama ^{NL} baseline. This is surprising, because it indicates that we can obtain both better performance and better compression by changing the tokenizer of a pre-trained LLM. Therefore, we conclude that *if Code Llama had changed its tokenizer before fine-tuning, it would have had a negligible impact on downstream performance, but a large positive impact on compression and inference speed.*

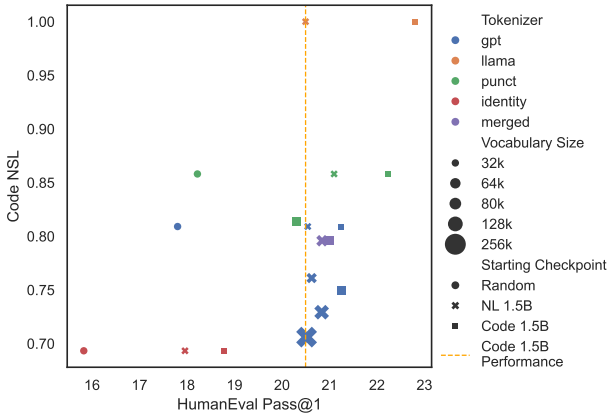

Figure 5. **Performance vs Code NSL**. We plot the HumanEval Pass@1 performance against Code NSL for our 1.5B LLMs fine-tuned with different base models and tokenizers. The x-axis is the downstream performance on code (larger is better) and the y-axis is the code NSL (smaller is better). The size of each point represents the vocabulary size of the tokenizer. Ideally, a model would be in the bottom-right quadrant and with a small vocabulary.

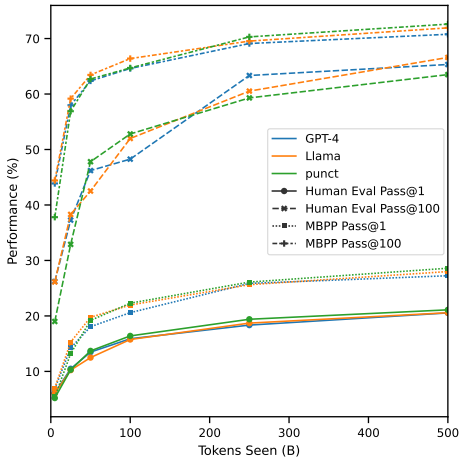

Figure 6. Performance of GPT-4 ^{NL}, Punct ^{NL} and Llama ^{NL} 1.5B at varying dataset sizes (5B, 25B, 50B, 100B, 250B, 500B). This figure demonstrates the impact of the number of tokens seen during training on end-task performance for models where the tokenizer is changed. We find both GPT-4 ^{NL} and Punct ^{NL} competitive with a LLM where the tokenizer is kept the same (Llama ^{NL}), after 50B tokens seen during training.

3.1. How much data?

It is clear that changing a tokenizer has large effect on the weights, and crucially that not every fine-tuning regime would be appropriate for a tokenizer change. Specifically, we want to measure how many tokens does it take for a LLM to recover performance on end-tasks after suffering from a switch in tokenizer.

We fine-tune the GPT-4 ^{NL}, Punct ^{NL} and Llama ^{NL} 32k 1.5B models on different subset sizes of the data (5B, 25B, 50B , 100B, 250B, 500B) and measure downstream performance. We keep all hyper-parameters and only adjust the learning rate schedule to fit the reduced datasets. We report our results in Figure 6. While we find a large difference between Punct ^{NL} and Llama ^{NL} when trained on only 5B tokens, this difference almost disappears and even inverts (on Pass@1) after 50B tokens. We therefore can only recommend that tokenizers are fine-tuned in regimes where there is enough training data for the model to adapt to the new distribution.

3.2. Influence of tokenizer size

	Human Eval		MBPP	
	Pass@1	**Pass@100**	**Pass@1**	**Pass@100**
32k	20.5%	65.3%	27.2%	70.8%
64k	20.6%	67.2%	27.6%	70.3%
128k	20.8%	64.2%	27.5%	70.8%
256k	20.5%	63.5%	27.6%	71.2%

Table 3. **Downstream performance GPT-4 ^{NL} 1.5B model depending on tokenizer size.**

We test the hypothesis of whether larger vocabulary sizes decrease downstream performance. As shown in Table 3, we change the tokenizer of a NL 1.5B model to that of a GPT-4 tokenizer of sizes 32k, 64k, 128k, and 256k. Calculating the Pearson correlation coefficient between the tokenizer vocabulary size and HumanEval Pass@1, results in a correlation coefficient of -0.13 with a p-value of 0.87. This indicates a very weak inverse relationship between vocabulary size and HumanEval Pass@1, but the high p-value indicates that this correlation is far from statistically significant. Therefore, we do not find that the vocabulary size (at this magnitude) to have an impact on end-goal performance and *reject the hypothesis that larger vocabulary sizes decrease task performance*.

3.3. Tokenizer update methods

In this section, we compare two methods to update the tokenizer of a pre-trained model: using Fast Vocabulary Transfer (FVT) and extending an existing tokenizer. In Appendix J, we also experiment with updating only the embedding and output weights, however this does not lead to improvements over full fine-tuning.

3.3.1. Vocabulary Transfer

Techniques such as Vocabulary Initialization with Partial Inheritance (VIPI) (Mosin et al., 2023) and Fast Vocabulary Transfer (FVT) (Gee et al., 2022) have been proposed to adapt the embedding space of pre-trained models by mapping a new tokenizer onto an existing vocabulary. VIPI and

	Human Eval		MBPP	
	Pass@1	Pass@100	Pass@1	Pass@100
GPT-4 ^{NL}	20.5%	65.3%	27.2%	70.8%
No-FVT GPT-4 ^{NL}	18.4%	58.6%	25.0%	69.2%
Merged ^{NL}	20.8%	67.5%	27.6%	70.9%

Table 4. We compare the performance of GPT-4 ^{NL} 32k with FVT and without FVT (No-FVT), and that of the extended Llama tokenizer (Merged).

FVT both act as a simple way to map the old embedding space onto the new, but still require a fine-tuning stage to align the new representations with the model. Gee et al. (2022) adapt a BERT base model (Devlin et al., 2019) with FVT using in-domain (medical, legal, and news) tokenizers to obtain efficiency gains at little cost to performance. Since, we use FVT to initialize our embedding matrix, we ablate FVT in Table 4. We confirm that FVT leads to noticeable improvement on all downstream tasks.

3.3.2. Tokenizer Extension

As an alternative to FVT, we study extending a tokenizer (e.g. the Llama tokenizer) by adding domain-specific tokens. As the extended tokenizer contains all the tokens of the previous tokenizer, it may make the transition smoother and have less impact on the performance of the model.

We train a 64k vocabulary tokenizer using Sentencepiece on code data, changing the pre-tokenization scheme to allow tokens over tabs and new line characters, and filter out tokens disallowed by the GPT-4 regular expression. We combine this in-domain tokenizer with the Llama tokenizer to obtain a tokenizer of size 80k which we refer to as Merged (see Table 1 for compression statistics). Table 4 reports the Merged ^{NL} results. We observe only small gains from starting with the Merged tokenizer compared to starting from an entirely distinct tokenizer such as GPT-4.

3.4. 7B models

	Human Eval		MBPP	
	Pass@1	Pass@100	Pass@1	Pass@100
Code Llama	32.1%	84.2%	41.9%	81.8%
Punct	30.4%	85.1%	41.4%	81.9%
GPT-4	30.8%	86.2%	42.6%	83.4%
Merged	31.2%	86.3%	42.0%	80.9%

Table 5. **Downstream performance of fine-tuned Llama 2 7Bs**. We report the Pass@1 performance on HumanEval and MBPP of three Llama 2 7B models after 500B additional tokens of code pre-training. We also report the Code Llama 7B model as a baseline.

Lastly, we fine-tune three 7B Llama models for 500B tokens and show that, as in the 1.5B LLMs, changing the tokenizer does not significantly impact code-generation performance. Table 5 shows the result of changing the original Llama tokenizer in the Llama 2 7B model from Touvron et al. (2023b) to Punct, GPT-4, and Merged tokenizers of vocabulary size 80k. Note that we opt to test for a larger vocabulary size as we showed in Section 2.5.1 that larger models such as a 7B can trade-off additional parameters for increased code compression. We compare our models to the 7B Code Llama model from Rozière et al. (2023), which used the tokenizer of Llama. Our results support our thesis that, with long-enough fine-tuning, tokenizers can be changed without sacrificing performance.

4. Related Works

Tokenizers in Machine Translation Tokenization has been a significant area of interest in multilingual tasks, such as Machine Translation, as different tokenization schemes can markedly influence model performance. Sennrich et al. (2016) introduced BPE for tokenization and were the first to use sub-word tokenization a solution for encoding rare or unseen words at test time. Rust et al. (2021) analyze the implications of multilingual tokenization schemes and find that specialized monolingual tokenizers outperform multilingual versions. More recently, Liang et al. (2023) demonstrate that expanding tokenizer vocabulary size and allocating a specific token quota for each language can substantially enhance performance in large-scale multilingual tasks. Liang et al. (2023) also argue that increasing the tokenizer size can be an efficient way to increase the number of trainable parameters, since only a fraction of the embedding matrix is used for a given input.

Tokenizers in Code Generation In the field of code generation, most LLMs follow standard tokenizer hyperparameters. Models like SantaCoder (Allal et al., 2023) and InCoder (Fried et al., 2023) train a 50k vocabulary tokenizer on code data. Fine-tuned code models derived from natural language LLMs, such as Codex (Chen et al., 2021), CodeGeeX (Zheng et al., 2023), and CodeGen (Nijkamp et al., 2023), do not update their tokenizers but extend existing ones like GPT-2's[1]. Code Llama (Rozière et al., 2023) fine-tunes Llama 2 (Touvron et al., 2023b) keeping the original 32k Llama tokenizer (Touvron et al., 2023a). Chirkova & Troshin (2023) find that custom code-tokenization can compress sequences by up to 17% without sacrificing performance on a PLBART (Ahmad et al., 2021) model.

5. Conclusion

This paper presents a comprehensive analysis of the impact of tokenization in modern LLMs. Our study reveals

[1]See Appendix C for an overview of tokenizer re-use in code generation.

that varying the size, pre-tokenization regular expressions, and training data of tokenizers can significantly impact the compression rate of the tokenizer. We showed that BPE tokenizers can compress code sequences by more than 40% compared to the Llama tokenizer, and more than 25% without any performance degradation (using GPT-4 256k). On top of that, we show that we can modify the tokenizer of a pre-trained LLM during fine-tuning if training for long enough (> 50B tokens). For models such as Code Llama, this allows for substantial gains in generation speed and effective context size, at no cost in performance.

We present recommendations to train efficient tokenizers, evaluating the effects of data distribution, vocabulary size and pre-tokenization scheme on the compression rates of tokenizers. We find that vocabulary size has little impact on coding performance, and present methods to find an optimal vocabulary size to optimize either memory or inference speed. For most use cases, we validate the use of the GPT-4 pre-tokenization regular expression, which strikes a good balance between compression and performance. Its downstream performance is similar to that of the Llama tokenizer, and close to Punct on coding benchmarks while bringing clear improvements in compression. We find that skipping pre-tokenization (Identity) can maximize compression at significant cost to performance. However, methods involving large-scale sampling and more concerned with the pass@100 than pass@1, or requiring large context sizes, might make good use of the maximal-compression Identity pre-tokenization. More broadly, although Punct sometimes outperforms GPT-4, we ultimately recommend using the GPT-4 as it provides an additional 5% compression. Finally, we hope this our results pushes researchers and practitioners to think further about the design of their tokenizer and consider changing to an in-domain tokenizer when fine-tuning.

Acknowledgements

We thank Fabian Gloeckle, Jonas Gehring, Nicolas Usunier and the other members of the CodeGen team at Meta for their helpful technical assistance throughout this project.

This work was supported in part by the UKRI Centre for Doctoral Training in Natural Language Processing, funded by the UKRI (grant EPS0224811) at the University of Edinburgh, School of Informatics and School of Philosophy, Psychology & Language Sciences and by the UKRI-funded TAS Governance Node (grant number EPV0266071).

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning and to encourage tokenizer optimizations for model efficiency. It could help make LLM inference faster and cheaper, resulting in broader LLM usage.

References

01.AI. Yi series models: Large language models. Hugging Face Model Repository, 2023. URL https://huggingface.co/01-ai/Yi-6B. Accessed: 17/11/2023.

Ahmad, W., Chakraborty, S., Ray, B., and Chang, K.-W. Unified pre-training for program understanding and generation. In Toutanova, K., Rumshisky, A., Zettlemoyer, L., Hakkani-Tur, D., Beltagy, I., Bethard, S., Cotterell, R., Chakraborty, T., and Zhou, Y. (eds.), *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 2655–2668, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.211. URL https://aclanthology.org/2021.naacl-main.211.

Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebron, F., and Sanghai, S. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In Bouamor, H., Pino, J., and Bali, K. (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 4895–4901, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.298. URL https://aclanthology.org/2023.emnlp-main.298.

Allal, L. B., Li, R., Kocetkov, D., Mou, C., Akiki, C., Ferrandis, C. M., Muennighoff, N., Mishra, M., Gu, A., Dey, M., Umapathi, L. K., Anderson, C. J., Zi, Y., Lamy-Poirier, J., Schoelkopf, H., Troshin, S., Abulkhanov, D., Romero, M., Lappert, M., Toni, F. D., del Río, B. G., Liu, Q., Bose, S., Bhattacharyya, U., Zhuo, T. Y., Yu, I., Villegas, P., Zocca, M., Mangrulkar, S., Lansky, D., Nguyen, H., Contractor, D., Villa, L., Li, J., Bahdanau, D., Jernite, Y., Hughes, S., Fried, D., Guha, A., de Vries, H., and von Werra, L. Santacoder: don't reach for the stars! *CoRR*, abs/2301.03988, 2023. doi: 10.48550/ARXIV.2301.03988. URL https://doi.org/10.48550/arXiv.2301.03988.

Almazrouei, E., Alobeidli, H., Alshamsi, A., Cappelli, A., Cojocaru, R., Debbah, M., Étienne Goffinet, Hesslow, D., Launay, J., Malartic, Q., Mazzotta, D., Noune, B., Pannier, B., and Penedo, G. The falcon series of open language models, 2023.

Anthropic. Introducing claude. Anthropic Blog, March 2023. URL https://www.anthropic.com/index/introducing-claude. Accessed: 17/11/2023.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al.

Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Biderman, S., Schoelkopf, H., Anthony, Q. G., Bradley, H., O'Brien, K., Hallahan, E., Khan, M. A., Purohit, S., Prashanth, U. S., Raff, E., et al. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*, pp. 2397–2430. PMLR, 2023.

Black, S., Biderman, S., Hallahan, E., Anthony, Q., Gao, L., Golding, L., He, H., Leahy, C., McDonell, K., Phang, J., Pieler, M., Prashanth, U. S., Purohit, S., Reynolds, L., Tow, J., Wang, B., and Weinbach, S. GPT-NeoX-20B: An open-source autoregressive language model. In Fan, A., Ilic, S., Wolf, T., and Gallé, M. (eds.), *Proceedings of BigScience Episode #5 – Workshop on Challenges & Perspectives in Creating Large Language Models*, pp. 95–136, virtual+Dublin, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.bigscience-1.9. URL https://aclanthology.org/2022.bigscience-1.9.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

Chirkova, N. and Troshin, S. Codebpe: Investigating subtokenization options for large language model pretraining on source code. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/pdf?id=htL4UZ344nF.

Deci. Introducing decicoder: The new gold standard in efficient and accurate code generation, August 2023. URL https://deci.ai/blog/. Accessed: 2024-01-18.

DeepSeek AI. Deepseek coder: A series of code language models. DeepSeek Coder Official Website, 2023. URL https://deepseekcoder.github.io/. Accessed: 17/11/2023.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In Burstein, J., Doran, C., and Solorio, T. (eds.), *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423.

Elsen, E., Odena, A., Nye, M., Taşırlar, S., Dao, T., Hawthorne, C., Moparthi, D., and Somani, A. Releasing Persimmon-8B, 2023. URL https://www.adept.ai/blog/persimmon-8b.

Forsythe, A. Tokenmonster: Ungreedy subword tokenizer and vocabulary trainer for python, go and javascript. https://github.com/alasdairforsythe/tokenmonster, 2023.

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, S., Zettlemoyer, L., and Lewis, M. Incoder: A generative model for code infilling and synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/pdf?id=hQwb-lbM6EL.

Gee, L., Zugarini, A., Rigutini, L., and Torroni, P. Fast vocabulary transfer for language model compression. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 409–416, Abu Dhabi, UAE, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-industry.41. URL https://aclanthology.org/2022.emnlp-industry.41.

Gee, L., Rigutini, L., Ernandes, M., and Zugarini, A. Multi-word tokenization for sequence compression. In Wang, M. and Zitouni, I. (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 612–621, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-industry.58. URL https://aclanthology.org/2023.emnlp-industry.58.

Gemma Team, G. T., Mesnard, T., Hardin, C., Dadashi, R., Bhupatiraju, S., Pathak, S., Sifre, L., Rivière, M., Kale, M. S., Love, J., Tafti, P., Hussenot, L., Sessa, P. G., Chowdhery, A., Roberts, A., Barua, A., Botev, A., Castro-Ros, A., Slone, A., Héliou, A., Tacchetti, A., Bulanova, A., Paterson, A., Tsai, B., Shahriari, B., Lan, C. L., Choquette-Choo, C. A., Crepy, C., Cer, D., Ippolito, D., Reid, D., Buchatskaya, E., Ni, E., Noland,

E., Yan, G., Tucker, G., Muraru, G.-C., Rozhdestvenskiy, G., Michalewski, H., Tenney, I., Grishchenko, I., Austin, J., Keeling, J., Labanowski, J., Lespiau, J.-B., Stanway, J., Brennan, J., Chen, J., Ferret, J., Chiu, J., Mao-Jones, J., Lee, K., Yu, K., Millican, K., Sjoesund, L. L., Lee, L., Dixon, L., Reid, M., Mikuła, M., Wirth, M., Sharman, M., Chinaev, N., Thain, N., Bachem, O., Chang, O., Wahltinez, O., Bailey, P., Michel, P., Yotov, P., Chaabouni, R., Comanescu, R., Jana, R., Anil, R., McIlroy, R., Liu, R., Mullins, R., Smith, S. L., Borgeaud, S., Girgin, S., Douglas, S., Pandya, S., Shakeri, S., De, S., Klimenko, T., Hennigan, T., Feinberg, V., Stokowiec, W., hui Chen, Y., Ahmed, Z., Gong, Z., Warkentin, T., Peran, L., Giang, M., Farabet, C., Vinyals, O., Dean, J., Kavukcuoglu, K., Hassabis, D., Ghahramani, Z., Eck, D., Barral, J., Pereira, F., Collins, E., Joulin, A., Fiedel, N., Senter, E., Andreev, A., and Kenealy, K. Gemma: Open models based on gemini research and technology, 2024.

Gowda, T. and May, J. Finding the optimal vocabulary size for neural machine translation. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 3955–3964, Online, Nov 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.352. URL https://aclanthology.org/2020.findings-emnlp.352.

Goyal, S., Ji, Z., Rawat, A. S., Menon, A. K., Kumar, S., and Nagarajan, V. Think before you speak: Training language models with pause tokens. *CoRR*, abs/2310.02226, 2023. doi: 10.48550/ARXIV.2310.02226. URL https://doi.org/10.48550/arXiv.2310.02226.

guidance-ai. A guidance language for controlling large language models. https://github.com/guidance-ai/guidance, 2023. Accessed: 2024-01-15.

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., de Las Casas, D., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., Lavaud, L. R., Lachaux, M., Stock, P., Scao, T. L., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. Mistral 7b. *CoRR*, abs/2310.06825, 2023. doi: 10.48550/ARXIV.2310.06825. URL https://doi.org/10.48550/arXiv.2310.06825.

Kocetkov, D., Li, R., Allal, L. B., Li, J., Mou, C., Ferrandis, C. M., Jernite, Y., Mitchell, M., Hughes, S., Wolf, T., Bahdanau, D., von Werra, L., and de Vries, H. The stack: 3 TB of permissively licensed source code. *CoRR*, abs/2211.15533, 2022. doi: 10.48550/ARXIV.2211.15533. URL https://doi.org/10.48550/arXiv.2211.15533.

Kudo, T. Subword regularization: Improving neural network translation models with multiple subword candidates. In Gurevych, I. and Miyao, Y. (eds.), *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, pp. 66–75. Association for Computational Linguistics, 2018. doi: 10.18653/V1/P18-1007. URL https://aclanthology.org/P18-1007/.

Kudo, T. and Richardson, J. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In Blanco, E. and Lu, W. (eds.), *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, EMNLP 2018: System Demonstrations, Brussels, Belgium, October 31 - November 4, 2018*, pp. 66–71. Association for Computational Linguistics, 2018. doi: 10.18653/V1/D18-2012. URL https://doi.org/10.18653/v1/d18-2012.

Kulal, S., Pasupat, P., Chandra, K., Lee, M., Padon, O., Aiken, A., and Liang, P. Spoc: Search-based pseudocode to code. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 11883–11894, 2019. URL https://arxiv.org/abs/1906.04908.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., Liu, Q., Zheltonozhskii, E., Zhuo, T. Y., Wang, T., Dehaene, O., Davaadorj, M., Lamy-Poirier, J., Monteiro, J., Shliazhko, O., Gontier, N., Meade, N., Zebaze, A., Yee, M., Umapathi, L. K., Zhu, J., Lipkin, B., Oblokulov, M., Wang, Z., V, R. M., Stillerman, J., Patel, S. S., Abulkhanov, D., Zocca, M., Dey, M., Zhang, Z., Moustafa-Fahmy, N., Bhattacharyya, U., Yu, W., Singh, S., Luccioni, S., Villegas, P., Kunakov, M., Zhdanov, F., Romero, M., Lee, T., Timor, N., Ding, J., Schlesinger, C., Schoelkopf, H., Ebert, J., Dao, T., Mishra, M., Gu, A., Robinson, J., Anderson, C. J., Dolan-Gavitt, B., Contractor, D., Reddy, S., Fried, D., Bahdanau, D., Jernite, Y., Ferrandis, C. M., Hughes, S., Wolf, T., Guha, A., von Werra, L., and de Vries, H. Starcoder: may the source be with you! *CoRR*, abs/2305.06161, 2023. doi: 10.48550/ARXIV.2305.06161. URL https://doi.org/10.48550/arXiv.2305.06161.

Liang, D., Gonen, H., Mao, Y., Hou, R., Goyal, N., Ghazvininejad, M., Zettlemoyer, L., and Khabsa, M. XLM-V: overcoming the vocabulary bottleneck in multilingual masked language models. In Bouamor, H., Pino, J., and Bali, K. (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pp.

13142–13152. Association for Computational Linguistics, 2023. URL https://aclanthology.org/2023.emnlp-main.813.

Limisiewicz, T., Balhar, J., and Mareček, D. Tokenization impacts multilingual language modeling: Assessing vocabulary allocation and overlap across languages. In Rogers, A., Boyd-Graber, J., and Okazaki, N. (eds.), *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 5661–5681, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-acl.350. URL https://aclanthology.org/2023.findings-acl.350.

Llama 3 team, A. S., Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Kelsey, A., Gangidi, A., Al-Dahle, A., Goldstand, A., Letman, A., Menon, A., Mathur, A., Schelten, A., Vaughan, A., Yang, A., Lupu, A., Alvarado, A., Gallagher, A., Gu, A., Ho, A., Poulton, A., Ryan, A., Fan, A., Ramchandani, A., Hartshorn, A., Mitra, A., Sravankumar, A., Korenev, A., Rao, A., Gabriel, A., Bharambe, A., Eisenman, A., Zhang, A., Rodriguez, A., Gregerson, A., Spataru, A., Roziere, B., Maurer, B., Leonhardi, B., Huang, B., Paranjape, B., Liu, B., Tang, B., Chern, B., Stojkovic, B., Fuller, B., Arenas, C. M., Zhou, C., Caucheteux, C., Nayak, C., Chu, C.-H., Bi, C., Cai, C., Cox, C., Marra, C., McConnell, C., Keller, C., Feichtenhofer, C., Touret, C., Wu, C., Wong, C., Ferrer, C. C., Allonsius, D., Kreymer, D., Haziza, D., Li, D., Pintz, D., Livshits, D., Wyatt, D., Adkins, D., Esiobu, D., Xu, D., Testuggine, D., David, D., Parikh, D., Choudhary, D., Mahajan, D., Liskovich, D., Garcia-Olano, D., Perino, D., Hupkes, D., Wang, D., Holland, D., Lakomkin, E., Lobanova, E., Tan, X. E., Dinan, E., Smith, E., Brinkman, E., Arcaute, E., Radenovic, F., Ozgenel, F., Caggioni, F., Seide, F., Zhang, F., Synnaeve, G., Schwarz, G., Lee, G., Badeer, G., Anderson, G., Nail, G., Mialon, G., Pang, G., Cucurell, G., Nguyen, H., Korevaar, H., Wang, H., Habeeb, H., Rudolph, H., Aspegren, H., Xu, H., Touvron, H., Kozlowska, I., Molybog, I., Tufanov, I., Zarov, I., Ibarra, I. A., Veliche, I.-E., Kloumann, I., Misra, I., Evtimov, I., Xu, J., Copet, J., Weissman, J., Geffert, J., Vranes, J., Asher, J., Park, J., Mahadeokar, J., Gaya, J.-B., Shah, J., van der Linde, J., Chan, J., Hong, J., Lee, J., Fu, J., Teboul, J., Chi, J., Huang, J., Wang, J., Yu, J., Bitton, J., Spisak, J., Pineau, J., Carvill, J., Park, J., Rocca, J., Johnstun, J., Jia, J., Alwala, K. V., U, K. H., Plawiak, K., Upasani, K., Veeraraghavan, K., Li, K., Heafield, K., Stone, K., El-Arini, K., Iyer, K., Malik, K., Chiu, K., Bhalla, K., Huang, K., Garg, L., Rantala-Yeary, L., van der Maaten, L., Chen, L., Silva, L., Bell, L., Zhang, L., Tan, L., Martin, L., Madaan, L., Wehrstedt, L., Blecher, L., de Oliveira, L., Muzzi, M., Khabsa, M., Avlani, M., Singh, M., Paluri, M., Zuckerberg, M., Kardas, M., Mankus, M., Oldham, M., Rita, M., Lennie, M., Pavlova, M., Keneally, M., Kambadur, M., Patel, M., Samvelyan, M., Clark, M., Lewis, M., Si, M., Singh, M. K., Metanat, M., Hassan, M., Goyal, N., Torabi, N., Usunier, N., Bashlykov, N., Bogoychev, N., Chatterji, N., Dong, N., Yang, O. A., Duchenne, O., Celebi, O., Parekh, P., Alrassy, P., Saab, P., Balaji, P., Rittner, P., Zhang, P., Li, P., Vasic, P., Weng, P., Zvyagina, P., Bhargava, P., Dubal, P., Krishnan, P., Koura, P. S., He, Q., Rodriguez, R., Srinivasan, R., Mitra, R., Calderer, R., Li, R., Stojnic, R., Raileanu, R., Battey, R., Wang, R., Girdhar, R., Patel, R., Sauvestre, R., Polidoro, R., Sumbaly, R., Taylor, R., Silva, R., Hou, R., Wang, R., Howes, R., Rinott, R., Hosseini, S., Bondu, S. J., Datta, S., Singh, S., Chugh, S., Dhillon, S., Pan, S., Bell, S., Edunov, S., Nie, S., Narang, S., Raparthy, S., Lindsay, S., Feng, S., Shen, S., Lin, S., Shankar, S., Bhosale, S., Zhang, S., Vandenhende, S., Wang, S., Kim, S. S., Batra, S., Sootla, S., Kehoe, S., Gururangan, S., Gupta, S., Virk, S., Borodinsky, S., Glaser, T., Herman, T., Best, T., Fowler, T., Georgiou, T., Scialom, T., Li, T., Mihaylov, T., Xiao, T., Karn, U., Goswami, V., Gupta, V., Ramanathan, V., Kerkez, V., Kumar, V. S., Gonguet, V., Vogeti, V., Poenaru, V., Mihailescu, V. T., Petrovic, V., Ivanov, V., Li, W., Chu, W., Xiong, W., Fu, W., Bouaziz, W., Meers, W., Constable, W., Martinet, X., Wu, X., Gao, X., Xie, X., Jia, X., Goldschlag, Y., LeCun, Y., Gaur, Y., Babaei, Y., Qi, Y., Li, Y., Wen, Y., Song, Y., Nam, Y., Hao, Y., Zhang, Y., Wang, Y., Mao, Y., He, Y., Coudert, Z. D., DeVito, Z., Hankir, Z., Wen, Z., Yan, Z., Chen, Z., Yang, Z., and Papakipos, Z. Llama 3. Llama blog, April 2024. URL https://llama.meta.com/llama3/. Accessed: 14/05/2024.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

MosaicML. Mpt-7b-instruct: A model for short-form instruction following. MosaicML Blog, May 2023. URL https://www.mosaicml.com/blog/mpt-7b. Accessed: 17/11/2023.

Mosin, V. D., Samenko, I., Kozlovskii, B., Tikhonov, A., and Yamshchikov, I. P. Fine-tuning transformers: Vocabulary transfer. *Artif. Intell.*, 317:103860, 2023. doi: 10.1016/J.ARTINT.2023.103860. URL https://doi.org/10.1016/j.artint.2023.103860.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., and Xiong, C. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda,*

May 1-5, 2023. OpenReview.net, 2023. URL https://openreview.net/pdf?id=iaYcJKpY2B_.

Nogueira, R. F., Jiang, Z., and Lin, J. Investigating the limitations of the transformers with simple arithmetic tasks. *CoRR*, abs/2102.13019, 2021. URL https://arxiv.org/abs/2102.13019.

OpenAI. GPT-4 technical report. *CoRR*, abs/2303.08774, 2023. doi: 10.48550/ARXIV.2303.08774. URL https://doi.org/10.48550/arXiv.2303.08774.

Provilkov, I., Emelianenko, D., and Voita, E. Bpe-dropout: Simple and effective subword regularization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pp. 1882–1892, Online, Jul 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.acl-main.170. URL https://aclanthology.org/2020.acl-main.170.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. 2019. URL https://api.semanticscholar.org/CorpusID:160025533.

Replit. Replit's new ai model now available on hugging face. Replit Blog, October 2023. URL https://blog.replit.com/replit-code-v1_5. Accessed: 17/11/2023.

Rozière, B., Lachaux, M., Chanussot, L., and Lample, G. Unsupervised translation of programming languages. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Canton-Ferrer, C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950, 2023. doi: 10.48550/ARXIV.2308.12950. URL https://doi.org/10.48550/arXiv.2308.12950.

Rust, P., Pfeiffer, J., Vulić, I., Ruder, S., and Gurevych, I. How good is your tokenizer? on the monolingual performance of multilingual language models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 3118–3135, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.243. URL https://aclanthology.org/2021.acl-long.243.

Scao, T. L., Fan, A., Akiki, C., Pavlick, E., Ili'c, S., Hesslow, D., Castagn'e, R., Luccioni, A. S., Yvon, F., Gallé, M., Tow, J., Rush, A. M., Biderman, S. R., Webson, A., Ammanamanchi, P. S., Wang, T., Sagot, B., Muennighoff, N., del Moral, A. V., Ruwase, O., Bawden, R., Bekman, S., McMillan-Major, A., Beltagy, I., Nguyen, H., Saulnier, L., Tan, S., Suarez, P. O., Sanh, V., Laurenccon, H., Jernite, Y., Launay, J., Mitchell, M., Raffel, C., Gokaslan, A., Simhi, A., Etxabe, A. S., Aji, A. F., Alfassy, A., Rogers, A., Nitzav, A. K., Xu, C., Mou, C., Emezue, C. C., Klamm, C., Leong, C., van Strien, D. A., Adelani, D. I., Radev, D. R., Ponferrada, E. G., Levkovizh, E., Kim, E., Natan, E., Toni, F. D., Dupont, G., Kruszewski, G., Pistilli, G., ElSahar, H., Benyamina, H., Tran, H. T., Yu, I., Abdulmumin, I., Johnson, I., Gonzalez-Dios, I., de la Rosa, J., Chim, J., Dodge, J., Zhu, J., Chang, J., Frohberg, J., Tobing, J. L., Bhattacharjee, J., Almubarak, K., Chen, K., Lo, K., von Werra, L., Weber, L., Phan, L., Allal, L. B., Tanguy, L., Dey, M., Muñoz, M. R., Masoud, M., Grandury, M., vSavsko, M., Huang, M., Coavoux, M., Singh, M., Jiang, M. T.-J., Vu, M. C., Jauhar, M. A., Ghaleb, M., Subramani, N., Kassner, N., Khamis, N., Nguyen, O., Espejel, O., de Gibert, O., Villegas, P., Henderson, P., Colombo, P., Amuok, P., Lhoest, Q., Harliman, R., Bommasani, R., L'opez, R., Ribeiro, R., Osei, S., Pyysalo, S., Nagel, S., Bose, S., Muhammad, S. H., Sharma, S. S., Longpre, S., maieh Nikpoor, S., Silberberg, S., Pai, S., Zink, S., Torrent, T. T., Schick, T., Thrush, T., Danchev, V., Nikoulina, V., Laippala, V., Lepercq, V., Prabhu, V., Alyafeai, Z., Talat, Z., Raja, A., Heinzerling, B., Si, C., Salesky, E., Mielke, S. J., Lee, W. Y., Sharma, A., Santilli, A., Chaffin, A., Stiegler, A., Datta, D., Szczechla, E., Chhablani, G., Wang, H., Pandey, H., Strobelt, H., Fries, J. A., Rozen, J., Gao, L., Sutawika, L., Bari, M. S., Al-Shaibani, M. S., Manica, M., Nayak, N. V., Teehan, R., Albanie, S., Shen, S., Ben-David, S., Bach, S. H., Kim, T., Bers, T., Févry, T., Neeraj, T., Thakker, U., Raunak, V., Tang, X., Yong, Z.-X., Sun, Z., Brody, S., Uri, Y., Tojarieh, H., Roberts, A., Chung, H. W., Tae, J., Phang, J., Press, O., Li, C., Narayanan, D., Bourfoune, H., Casper, J., Rasley, J., Ryabinin, M., Mishra, M., Zhang, M., Shoeybi, M., Peyrounette, M., Patry, N., Tazi, N., Sanseviero, O., von Platen, P., Cornette, P., Lavall'ee, P. F., Lacroix, R., Rajbhandari, S., Gandhi, S., Smith, S., Requena, S., Patil, S., Dettmers, T., Baruwa,

A., Singh, A., Cheveleva, A., Ligozat, A.-L., Subramonian, A., N'ev'eol, A., Lovering, C., Garrette, D. H., Tunuguntla, D. R., Reiter, E., Taktasheva, E., Voloshina, E., Bogdanov, E., Winata, G. I., Schoelkopf, H., Kalo, J.-C., Novikova, J., Forde, J. Z., Tang, X., Kasai, J., Kawamura, K., Hazan, L., Carpuat, M., Clinciu, M., Kim, N., Cheng, N., Serikov, O., Antverg, O., van der Wal, O., Zhang, R., Zhang, R., Gehrmann, S., Mirkin, S., Pais, S. O., Shavrina, T., Scialom, T., Yun, T., Limisiewicz, T., Rieser, V., Protasov, V., Mikhailov, V., Pruksachatkun, Y., Belinkov, Y., Bamberger, Z., Kasner, Z., Kasner, Z., Pestana, A., Feizpour, A., Khan, A., Faranak, A., Santos, A. S. R., Hevia, A., Unldreaj, A., Aghagol, A., Abdollahi, A., Tammour, A., HajiHosseini, A., Behroozi, B., Ajibade, B. A., Saxena, B. K., Ferrandis, C. M., Contractor, D., Lansky, D. M., David, D., Kiela, D., Nguyen, D. A., Tan, E., Baylor, E., Ozoani, E., Mirza, F. T., Ononiwu, F., Rezanejad, H., Jones, H., Bhattacharya, I., Solaiman, I., Sedenko, I., Nejadgholi, I., Passmore, J., Seltzer, J., Sanz, J. B., Fort, K., Dutra, L., Samagaio, M., Elbadri, M., Mieskes, M., Gerchick, M., Akinlolu, M., McKenna, M., Qiu, M., Ghauri, M., Burynok, M., Abrar, N., Rajani, N., Elkott, N., Fahmy, N., Samuel, O., An, R., Kromann, R. P., Hao, R., Alizadeh, S., Shubber, S., Wang, S. L., Roy, S., Viguier, S., Le, T.-C., Oyebade, T., Le, T. N. H., Yang, Y., Nguyen, Z. K., Kashyap, A. R., Palasciano, A., Callahan, A., Shukla, A., Miranda-Escalada, A., Singh, A. K., Beilharz, B., Wang, B., de Brito, C. M. F., Zhou, C., Jain, C., Xu, C., Fourrier, C., Perin'an, D. L., Molano, D., Yu, D., Manjavacas, E., Barth, F., Fuhrimann, F., Altay, G., Bayrak, G., Burns, G., Vrabec, H. U., Bello, I. I., Dash, I., Kang, J. S., Giorgi, J., Golde, J., Posada, J. D., Sivaraman, K., Bulchandani, L., Liu, L., Shinzato, L., de Bykhovetz, M. H., Takeuchi, M., Pàmies, M., Castillo, M. A., Nezhurina, M., Sanger, M., Samwald, M., Cullan, M., Weinberg, M., Wolf, M., Mihaljcic, M., Liu, M., Freidank, M., Kang, M., Seelam, N., Dahlberg, N., Broad, N. M., Muellner, N., Fung, P., Haller, P., Chandrasekhar, R., Eisenberg, R., Martin, R., Canalli, R. L., Su, R., Su, R., Cahyawijaya, S., Garda, S., Deshmukh, S. S., Mishra, S., Kiblawi, S., Ott, S., Sang-aroonsiri, S., Kumar, S., Schweter, S., Bharati, S. P., Laud, T., Gigant, T., Kainuma, T., Kusa, W., Labrak, Y., Bajaj, Y., Venkatraman, Y., Xu, Y., Xu, Y., Xu, Y., Tan, Z. X., Xie, Z., Ye, Z., Bras, M., Belkada, Y., and Wolf, T. Bloom: A 176b-parameter open-access multilingual language model. *ArXiv*, abs/2211.05100, 2022. URL https://api.semanticscholar.org/CorpusID:253420279.

Sennrich, R., Haddow, B., and Birch, A. 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)*, pp. 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1162. URL https://aclanthology.org/P16-1162.

Thawani, A., Pujara, J., Ilievski, F., and Szekely, P. Representing numbers in NLP: a survey and a vision. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 644–656, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.53. URL https://aclanthology.org/2021.naacl-main.53.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023a. doi: 10.48550/ARXIV.2302.13971. URL https://doi.org/10.48550/arXiv.2302.13971.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Canton-Ferrer, C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023b. doi: 10.48550/ARXIV.2307.09288. URL https://doi.org/10.48550/arXiv.2307.09288.

Wang, Y., Wang, W., Joty, S., and Hoi, S. C. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, Online and Punta Cana, Dominican Republic, Nov 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.685. URL https://aclanthology.org/2021.emnlp-main.685.

Wenzek, G., Lachaux, M.-A., Conneau, A., Chaudhary, V., Guzmán, F., Joulin, A., and Grave, E. CCNet: Extracting high quality monolingual datasets from web crawl data. In Calzolari, N., Béchet, F., Blache, P., Choukri, K., Cieri, C., Declerck, T., Goggi, S., Isahara, H., Maegaard, B., Mariani, J., Mazo, H., Moreno, A.,

Odijk, J., and Piperidis, S. (eds.), *Proceedings of the Twelfth Language Resources and Evaluation Conference*, pp. 4003–4012, Marseille, France, May 2020. European Language Resources Association. ISBN 979-10-95546-34-4. URL https://aclanthology.org/2020.lrec-1.494.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., and Brew, J. Huggingface's transformers: State-of-the-art natural language processing. *CoRR*, abs/1910.03771, 2019. URL http://arxiv.org/abs/1910.03771.

Xue, L., Barua, A., Constant, N., Al-Rfou, R., Narang, S., Kale, M., Roberts, A., and Raffel, C. ByT5: Towards a token-free future with pre-trained byte-to-byte models. *Transactions of the Association for Computational Linguistics*, 10:291–306, 2022. doi: 10.1162/tacl_a_00461. URL https://aclanthology.org/2022.tacl-1.17.

Zheng, Q., Xia, X., Zou, X., Dong, Y., Wang, S., Xue, Y., Shen, L., Wang, Z., Wang, A., Li, Y., Su, T., Yang, Z., and Tang, J. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In Singh, A. K., Sun, Y., Akoglu, L., Gunopulos, D., Yan, X., Kumar, R., Ozcan, F., and Ye, J. (eds.), *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, KDD 2023, Long Beach, CA, USA, August 6-10, 2023*, pp. 5673–5684. ACM, 2023. doi: 10.1145/3580305.3599790. URL https://doi.org/10.1145/3580305.3599790.

Zipf, G. K. *Human Behavior and the Principle of Least Effort*. Addison-Wesley, 1949.

Zouhar, V., Meister, C., Gastaldi, J. L., Du, L., Sachan, M., and Cotterell, R. Tokenization and the noiseless channel. In Rogers, A., Boyd-Graber, J. L., and Okazaki, N. (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pp. 5184–5207. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.ACL-LONG.284. URL https://doi.org/10.18653/v1/2023.acl-long.284.

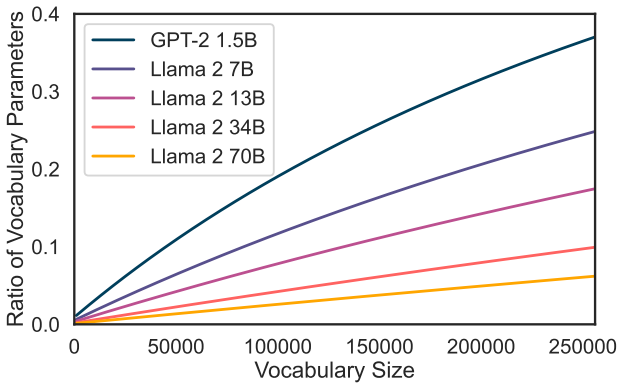

Figure 7. Proportion of parameters used by different vocabulary sizes for different model sizes of Llama 2 and GPT-2.

A. Compression Evaluation Dataset

Our English held-out dataset is 1000 examples from CCnet (Wenzek et al., 2020) and 1000 examples from the English subset of Wikipedia.

To create our multilingual evaluation set, we hold-out 1000 examples for each language present in the Wikipedia dataset. This constitutes of 1000 articles from the following languages: Italian, Bulgarian, Egyptian Arabic, Spanish, Serbian, German, Ukrainian, Arabic, Czech, Catalan, Slovenian, Persian, Polish, Russian, Finnish, Swedish, Portuguese, French, Japanese, Croatian, Chinese, Romanian, Dutch, Indonesian, Hungarian, Korean, Danish, and Vietnamese.

To create our code evaluation dataset, we hold-out 1000 files from the 30 most popular programming languages in the Stack dataset (Kocetkov et al., 2022): Assembly, Batchfile, C, C#, C++, CMake, CSS, Dockerfile, FORTRAN, GO, HTML, Haskell, Java, JavaScript, Julia, Lua, Makefile, Markdown, PHP, Perl, PowerShell, Python, Ruby, Rust, SQL, Scala, Shell, TeX, TypeScript, and Visual Basic.

B. Vocabulary Size and Model Size

We show the proportion of parameters used by tokenizers of different vocabulary sizes for different model sizes of Llama 2 and GPT-2 for scale in Figure 7. This shows that expanding the number of parameters in a LLM of size 70B only marginally increases the total size of the model and the ratio of vocab parameters to model parameters. While increasing the vocabulary size for smaller models, such as a GPT-2 XL 1.5B model, has a much larger effect on the total number of parameters.

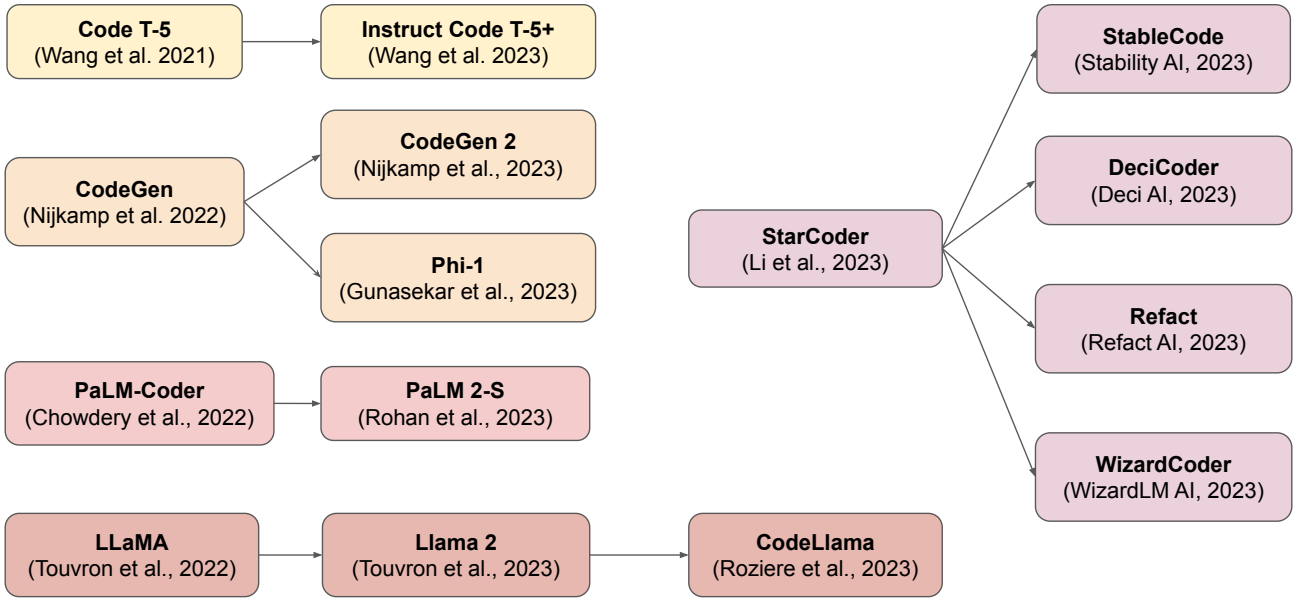

Figure 8. **Tokenizer re-use is common across code generation LLMs.**

C. Tokenizer Re-use

Figure 8 shows examples of tokenizer re-use across multiple SOTA code generation models. This is not a comprehensive mapping, but serves to illustrate how often tokenization is left as an implementation detail that is rarely innovated upon.

D. Normalization

Normalization is a pre-tokenization step that aims to reduce the complexity of text data by converting it into a unified format. Different forms of UTF-8 normalizations have both been used as defaults by different tokenizer libraries – Sentencepiece (Kudo & Richardson, 2018) uses Normalization Form KC (NFKC) and TokenMonster (Forsythe, 2023) uses Normalization Form D (NFD).

- **NFKC Normalization**: In NFKC, characters are transformed to their most common, compatible form. For instance, a superscript UTF-8 number will get converted to a standard number: $\text{NFKC}(^2) = 2$
- **NFD Normalization**: Conversely, NFD decomposes characters into their basic components, separating letters from their diacritical marks. For example, a character with an accent would be split into its letter and accent components: $\text{NFD}(\tilde{n}) = n +\tilde{}$

Despite the potential advantages of a normalization step, these are generally not reversible transformations[2]. Therefore the tokens processed by the tokenizer may not retain their original form, violating $x = \text{encode}(\text{decode}(x))$. This can lead to sub-optimal normalization, where for instance converting the superscript ^{2} to its base representation 2 throws away the power semantic information – which could be crucial for mathematical reasoning. We forgo the use of any normalization methods in our proposed tokenizers and broadly recommend that modern tokenizers be reversible.

E. The Punct Tokenizer

In Punct, we remove the GPT-4 English-specific contractions and prevent certain white-spaces and punctuation tokens such as `\t` or `.` to be encoded at the start of a letter-only token. The Punct pre-tokenizer regular expression is shown in Figure 3, and, compared to GPT-4 trades away compression in exchange for composition. For instance, Punct prevents any form of tokens that starts with a `.` (period) which is a common shorthand to accessing attributes in many programming languages. Consider the `.append` method in python, the GPT-4 tokenizer would tokenize the entire sequence `.append` as a single token while Punct would tokenize `.` and `append` separately. We use Punct to test whether a stronger separation between syntax and semantics, simplifies the language generation task and translates to greater downstream performance.

In our initial results, shown in Table 2, we found Punct ^{NL} 32k to be better than both GPT-4 ^{NL} 32k and Llama ^{NL} on Pass@1 performance in both HumanEval and MBPP. How-

[2]NFD is mostly reversible if the input text is in form C but has rare edge cases.

ever, the results obtained on a 7B model (Table 5) using Punct 80k show no significant edge over the Code Llama baseline and either the Merged or GPT-4 80k tokenizers. We thus cannot conclude that a stronger separation between syntax and semantics leads to greater downstream performance. Instead, we would recommend usage of the GPT-4 regular expression tokenizer since it provides an additional 5% compression over Punct at no significant cost to performance.

F. Tokenizer Compression of Popular LLMs

Table 11 shows the tokenizer compression rates for a large number of popular LLMs. Most, but not all, of the models selected are models that have been promoted or advertised for code generation. We also include our own tokenizers for comparison.

G. Examples of tokenization

We compare different tokenizers on a sample code in Table 12. In our example snippet, we can see the effects of the different pre-tokenization schemes in practice. For instance, GPT-2 did not restrict the max number of consecutive digits in its pre-tokenization and therefore has a token to represent '1000', whereas the GPT-4 and Punct regular expression limit the maximum number of digits to 3, and therefore encode '1000' as '100' and '0'. Llama uses single digits and thus uses 4 tokens to encode the number '1000'. We also see that the GPT-4 pre-tokenization allows learning tokens such as '.append', which Punct cannot.

We can also observe the effects of having an unrestricted pre-tokenization scheme such as Identity, which even learn a token for the numpy import statement including the new line symbol. These types of savings allows Identity to save more than 50% tokens compared to Llama. However, we can also observe the downsides of the Identity tokenizer in its first tokenization of 'tqdm '. Since the BPE merge rule for 'm' and ' ' comes first, 'tqdm ' is tokenized as 'tq','d' and 'm ' separately. Whereas the second instance of 'tqdm', which is followed by a new line token, is able to be tokenized into a single token 'tqdm'.

H. Training NL and Code 1.5B Base Models

To test our hypotheses, we train two GPT-2 XL 1.5B base models (Radford et al., 2019) which we refer to as NL 1.5B and Code 1.5B. NL 1.5B is trained for 2T tokens on a general data-mix resembling that of Llama 2, while Code 1.5B is fine-tuned from NL 1.5B for 500B tokens using a code-specific data-mix. We use the same hyper-parameters as in Llama 2 (Touvron et al., 2023b) and Code Llama (Rozière et al., 2023), see Table 7 for an overview.

I. Rényi Entropy in Code Tokenizers

As shown in Table 11, following the methodology of Zouhar et al. (2023), we measure the Rényi entropy of all tokenizers with an $\alpha = 2.5$. Rényi entropy quantifies the evenness in the distribution of tokens, penalizing those distributions with highly uneven token frequencies. Zouhar et al. (2023) reported a correlation between Rényi entropy and downstream performance, as indicated by the BLEU score in a Machine Translation task.

However, our analysis with trained 1.5B models reveals a different trend. We found a significant negative correlation between Rényi entropy and Human Eval Pass@1 (Pearson $r = -0.7765, p = 0.040$) and MBPP Pass@1 (Pearson $r = -0.9007, p = 0.0057$). This suggests that a higher Rényi entropy ($\alpha = 2.5$), contrary to expectations, correlates with lower performance in these code generation metrics. Notably, the Identity tokenizer, which performed worst in our evaluations, showed the highest Rényi entropy. This can be attributed to the lack of a pre-tokenizer, allowing the BPE algorithm to more evenly distribute tokens according to frequency.

J. Freezing Weights

	Human Eval		MBPP	
	Pass@1	**Pass@100**	**Pass@1**	**Pass@100**
0B	22.2%	68.7%	29.8%	73.6%
100B	21.3%	67.5%	28.5%	72.3%
500B	17.8%	59.7%	24.5%	68.9%

Table 6. **Keeping LLM weights frozen for a number of tokens.** We compare the performance of Punct ^{NL} with weights frozen for different numbers of tokens and report results are on Pass@1 and Pass@100 for Human Evaluation and MBPP. The rows (0B, 100B, 500B) represent the number of tokens for which weights were frozen for. We find freezing weights ineffective to adapt a LLM to a change in its tokenizer.

Since the baseline model is assumed to be a pre-trained LLM, we also experiment with freezing the weights of the model that do not pertain to the vocabulary and training only the weights affected by the change – the embedding and output layers. We test two different setups where we 1) freeze the models weights except the embedding/output layers only for 100B tokens and then unfreeze the entire model for the remaining 400B tokens, and 2)freeze the model weights except the embedding/output layers for the 500B tokens. Our motivation for 1) is to test whether the model benefits from an initial alignment of its input/outputs to its intermediate representations, and for 2) is to test whether the model can learn to adapt its weights to the new input/outputs it receives without changing its internal structure.

Table 6 shows that only training only training the embedding and output weights for 100B (20% of training) or for 500B (100% of training) leads to worse performance on all downstream metrics. We find freezing weights to be an ineffective strategy to aligning the vocabulary-specific weights of the model with its internal structure.

K. Tricks for more robust Tokenizers

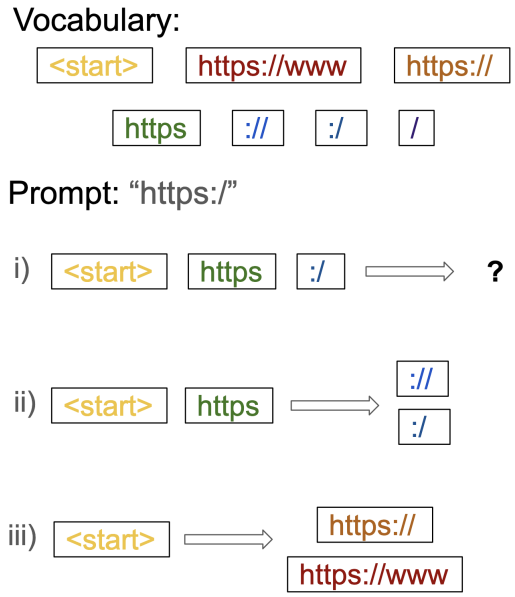

Figure 9. **Token healing strategies.** We consider three decoding strategies given the prompt "https:/". **i)** is the decoding case without token healing, where the next decoding step is unconstrained. **ii)** is a single token healing step, where the prompt is backtracked by a single token and a single constrained decoding step is done over the possible tokens that fit the observed text (":/"). **iii)** is the case with N backtracking steps, where the prompt is backtracked until there does not exist a token in the vocabulary which can cover the observation ("https:/"). A single constrained decoding step is then done over the possible tokens that fit the observation. The N steps token healing method can therefore lead to different decoding outcomes than the single step backtracking.

K.1. Token healing

	Human Eval		MBPP	
	Pass@1	Pass@100	Pass@1	Pass@100
Without token healing				
Llama ^{C}	22.6%	67.6%	30.0%	74.1%
GPT-4 ^{C}	21.4%	67.6%	28.8%	74.6%
Identity ^{C}	0.1%	3.5%	23.6%	68.8%
With token healing				
Llama ^{C}	22.8%	68.6%	30.1%	74.4%
GPT-4 ^{C}	21.2%	67.5%	28.9%	74.9%
Identity ^{C}	18.8%	66.1%	23.4%	67.8%

Table 8. **Token Healing Results.** We show results for three different tokenizers of size 32k trained from Code Llama for 500B tokens. For both Llama and GPT-4, token healing has very little impact on the evaluation metrics. Using the Identity tokenizer, the LLM is mostly unable to generate valid code HumanEval if not using token healing. This is likely because the prompt boundary in HumanEval splits a token learned in the Identity tokenizer, and thus disturbs the expected token distribution.

Token healing is a technique used to address biases introduced by tokenization at prompt boundaries (guidance-ai, 2023). This bias occurs when the prompt ends with a string that could be encoded as the beginning of a token, leading to a skewed distribution. Token healing backs up the generation process by one token before the end of the prompt and constrains the first token generated to have a prefix that matches the last token in the prompt to mitigate this bias. We find token healing to have a negligible effect on tokenizers with well-defined word/token boundaries, however token healing has a large impact on tokenizers such as the Identity tokenizer where alphanumeric tokens are allowed to include new lines and spaces (which are common separators in prompts).

See Figure 9 for an illustrated example of different token healing strategies. We implement a version of token healing that steps back N tokens until a valid token boundary is found. This is different from the original implementation of token healing (guidance-ai, 2023) that naively steps back by a single step. This is because some of the tokenizers we test, such as the identity tokenizer, can contain tokens that are entire sentences, and simply doing a single step back might not allow that sentence token to be recovered.

We also quantitatively assess the impact of token healing by running our evaluations with and without token healing. Our results shown in Table 8 indicate that token healing is particularly effective for tokenizers with less defined boundaries, such as the Identity tokenizer. In fact, the LLMs using the Identity tokenizer are unable to generate valid code given the prompt in HumanEval without token healing.

While token healing shows promise in addressing tokeniza-

Configuration	NL 1.5B	Code 1.5B
Initialization	Random	NL 1.5B
Number of Parameters	1.5B	1.5B
Number of Layers	48	48
Hidden Size	1600	1600
Number of Attention Heads	25	25
Sequence Length	4096	4096
Optimizer	AdamW (Loshchilov & Hutter, 2019)	AdamW (Loshchilov & Hutter, 2019)
Optimizer Parameters	$\beta_1 = 0.9, \beta_2 = 0.95$	$\beta_1 = 0.9, \beta_2 = 0.95$
Learning Rate	3×10^{-4}	3×10^{-4}
Learning Rate Schedule	Cosine	Cosine
Learning Rate Decay	$\frac{1}{10}lr$	$\frac{1}{30}lr$
Number of Tokens Seen	2T	500B
Data	General	Code-specific

Table 7. NL 1.5B Pre-Training configurations. Globally, we re-use the same parameters as Llama 2 (Touvron et al., 2023b) and Code Llama (Rozière et al., 2023) and only change the architecture to the smaller 1.5B GPT-2 XL model type.

tion bias, it is not without limitations. One key challenge is determining the optimal constrained decoding path for a given prompt. As we show in Figure 9, different numbers of backtracks might lead to different decoding outcomes. In fact, to truly approximate the text distribution, a more correct implementation of token healing would have to implement a constrained beam-search. One would have to backtrack by N steps and do multiple constrained decoding steps over the vocabulary to allow the smaller partial but potentially valid tokens to be generated. We leave further exploration of token healing and beam-search implementation as future work.

K.2. BPE dropout

Another trick that one can use to adapt to new tokenizers or make a model more robust in its tokenization scheme is BPE dropout (Provilkov et al., 2020). BPE dropout randomly drops merge entries in the BPE merge table, thus injecting noise in the tokenization process. This leads to sentences being encoded sub-optimally, but with more variation in the token distribution than would naturally be observed with greedy encoding. A model would therefore be able to see the same sentence being encoded in multiple different ways.

Provilkov et al. (2020) observe that BPE dropout improves MT performance, even in high-resource settings, and particularly when applied to noisy sources. Additionally, BPE dropout seems to produce more fine-grained segmentation, alleviate issues with frequent sequences of characters, and lead to better embedding spaces compared to traditional BPE. Ultimately, BPE dropout makes the LLM more robust to specific tokenization and to prompt boundaries where the tokens encoded might not reflect the observed token distribution.

L. Correlation between Code NSL and Downstream Task Performance

The following tables summarize the correlation results between Code NSL and performance on downstream tasks, with and without the identity tokenizer which is shown to skew the statistical analysis due to its poor performance metrics.

	Human Eval		MBPP	
	Pass@1	Pass@100	Pass@1	Pass@100
Pearson (r)	0.49	0.31	0.59	0.63
Pearson p	0.04	0.21	0.01	0.00
Spearman (ρ)	0.39	0.31	0.68	0.66
Spearman p	0.11	0.22	0.00	0.00

Table 9. Correlation results when including Identity tokenizer

	Human Eval		MBPP	
	Pass@1	Pass@100	Pass@1	Pass@100
Pearson (r)	0.21	0.08	0.33	0.39
Pearson p	0.45	0.78	0.23	0.15
Spearman (ρ)	0.05	0.06	0.45	0.42
Spearman p	0.86	0.84	0.09	0.12

Table 10. Correlation results when excluding Identity tokenizer

While we find Code NSL to have significant effects on MBPP performance in Table 9, we believe these results is misleading. These calculations include the Identity tokenizer which is maximally compressive but obtains poor performance compared to other tokenizers and skews the analysis.

In Table 10 we exclude the identity tokenizer from the analysis and inconclusive results. Therefore, similarly to the correlation between Vocabulary Size and HumanEval Pass@1,

we fail to find a significant effect between NSL and performance. Rather what we find is in practice, is that using a tokenizer such as GPT-4 256k, we are able to obtain a 25% increase in compression at no cost to downstream metrics.

A theoretical case could still be made that compression hurts performance as it limits the amount of effective FLOPs available during the generation process. A model that compresses more will have less steps to compute to obtain the same answer. We fail to find direct evidence of these dynamics in our experiments.

	Vocabulary Size	Normalized Sequence Length (↓)				Bytes per Token (↑)				Avg. Renyi ($\alpha = 2.5$)
		Avg.	Code	Eng.	Mult.	Avg.	Code	Eng.	Mult.	
ByT5 (Xue et al., 2022)	256	3.10	2.70	3.82	2.79	1.00	1.00	1.00	1.00	0.51
CodeT5 (Wang et al., 2021)	32k	1.29	0.94	1.11	1.83	2.69	2.89	3.45	1.72	0.44
GPT-2 (Radford et al., 2019)	50k	1.13	1.19	0.86	1.33	2.97	2.28	4.44	2.19	0.38
DeepSeek Coder (DeepSeek AI, 2023)	32k	1.06	1.00	0.98	1.19	3.00	2.70	3.89	2.40	0.45
CodeGen (Nijkamp et al., 2023)	50k	1.05	0.95	0.86	1.33	3.16	2.83	4.44	2.19	0.43
SantaCoder (Allal et al., 2023)	49k	1.04	0.88	1.07	1.17	3.01	3.08	3.59	2.36	0.47
Yi-6B (01.AI, 2023)	64k	1.03	1.00	0.93	1.16	3.07	2.70	4.10	2.43	0.43
InCoder (Fried et al., 2023)	50k	1.03	0.74	1.02	1.31	3.17	3.66	3.73	2.11	0.51
Falcon (Almazrouei et al., 2023)	65k	1.00	0.94	0.88	1.19	3.23	2.89	4.33	2.47	0.42
Persimmon (Elsen et al., 2023)	192k	1.00	0.95	0.74	1.32	3.43	2.84	5.19	2.24	0.34
Replit (Replit, 2023)	32k	1.00	0.85	1.06	1.10	3.10	3.19	3.60	2.51	0.48
Llama (Touvron et al., 2023a)	32k	1.00	1.00	1.00	1.00	3.10	2.70	3.82	2.79	0.46
Mistral (Jiang et al., 2023)	32k	0.99	0.99	0.97	1.02	3.13	2.74	3.93	2.72	0.45
Starcoder (Li et al., 2023)	49k	0.99	0.87	1.04	1.07	3.13	3.13	3.69	2.58	0.47
DeciCoder (Deci, 2023)	49k	0.99	0.87	1.04	1.07	3.13	3.13	3.69	2.58	0.47
Punct	32k	0.98	0.86	0.96	1.11	3.20	3.15	3.98	2.48	0.48
GPT-4	32k	0.97	0.81	0.97	1.13	3.24	3.35	3.95	2.44	0.50
gpt-neox (Black et al., 2022)	50k	0.93	0.90	0.86	1.02	3.37	2.98	4.42	2.71	0.45
Pythia (Biderman et al., 2023)	50k	0.93	0.90	0.86	1.02	3.37	2.98	4.42	2.71	0.45
MPT (MosaicML, 2023)	50k	0.93	0.90	0.86	1.02	3.37	2.98	4.42	2.71	0.45
Identity	32k	0.92	0.69	0.89	1.16	3.54	3.94	4.30	2.37	0.61
Claude (Anthropic, 2023)	65k	0.91	0.84	0.87	1.04	3.43	3.22	4.41	2.66	0.44
Punct	64k	0.90	0.82	0.89	0.99	3.45	3.29	4.28	2.79	0.46
Merged	80k	0.90	0.80	0.95	0.94	3.45	3.42	4.01	2.93	0.48
GPT-4	64k	0.89	0.76	0.90	1.01	3.52	3.55	4.27	2.72	0.49
Punct	80k	0.88	0.81	0.88	0.95	3.52	3.32	4.35	2.89	0.46
GPT-4	80k	0.87	0.75	0.88	0.98	3.59	3.61	4.35	2.82	0.48
Punct	100k	0.86	0.81	0.86	0.92	3.59	3.35	4.42	2.99	0.46
GPT-4 (OpenAI, 2023)	100k	0.85	0.75	0.84	0.95	3.68	3.59	4.54	2.92	0.47
GPT-4	100k	0.85	0.74	0.86	0.94	3.67	3.66	4.43	2.92	0.48
Punct	128k	0.85	0.80	0.85	0.89	3.66	3.38	4.49	3.10	0.45
GPT-4	128k	0.83	0.73	0.85	0.91	3.75	3.71	4.50	3.03	0.47
Gemma* (Gemma Team et al., 2024)	256k	0.83	0.89	0.86	0.73	3.79	3.04	4.43	3.90	0.43
Identity	64k	0.82	0.63	0.79	1.04	3.94	4.36	4.83	2.64	0.61
Bloom (Scao et al., 2022)	250k	0.81	0.77	0.84	0.81	3.91	3.49	4.56	3.67	0.46
Llama 3* (Llama 3 team et al., 2024)	128k	0.80	0.75	0.84	0.80	3.90	3.60	4.54	3.56	0.48
Identity	80k	0.80	0.61	0.76	1.01	4.07	4.50	5.00	2.71	0.61
GPT-4	256k	0.78	0.71	0.82	0.82	3.95	3.83	4.66	3.35	0.47
Identity	100k	0.77	0.59	0.74	0.98	4.20	4.63	5.16	2.80	0.61

Table 11. Compression metrics for tokenizers calculated on Code, English, and Multilingual datasets. Normalized Sequence Length is a ratio calculated against the original Llama tokenizer, and indicates the average sequence length that a tokenizer would have compared to Llama. Bytes per tokens measures the number of UTF-8 bytes per (tokenized) token, where a higher Bytes per token would imply a greater level of compression. Average Renyi (Zouhar et al., 2023) is a distribution metric that measures how evenly the tokenizer allocates its vocabulary for a given text, it is calculated here with $\alpha = 2.5$ (see discussion in Appendix I). Note that all of our models shown in this table have been trained on the same split of data (70% code and 30% English). * indicates tokenizers and models that were published after our work. Llama 3 was influenced by our research on efficient tokenizer training.

Original snippet	import numpy as np
	from tqdm import tqdm
	
	# digits: 1000
	l = []
	for i in range(1000):
	  l.append(str(i).zfill(3)))

GPT-2 (Radford et al., 2019) Encoded length: 52	\n import•numpy•as•np\n from•tqdm•import•tqdm\n \n #•digits:•1000•\n l•=•[]\n for•i•in•range(1000):\n ••••l.append(str(i).zfill(3)))\n
Llama (Touvron et al., 2023a) Encoded length: 57	•\n import•numpy•as•np\n from•tqdm•import•tqdm\n \n #•digits:•1000•\n l•=•[]\n for•i•in•range(1000):\n ••••l.append(str(i).zfill(3)))\n
GPT-4 (OpenAI, 2023) Encoded length: 40	\n import•numpy•as•np\n from•tqdm•import•tqdm\n\n #•digits:•1000•\n l•=•[]\n for•i•in•range(1000):\n ••••l.append(str(i).zfill(3)))\n
Identity 100k Encoded length: 25	\nimport•numpy•as•np \nfrom•tqdm•import•tqdm \n\n#•digits:•1000•\n l•=•[]\n for•i•in•range(1000):\n•••• l.append(str(i).zfill(3)))\n
GPT-4 100k Encoded length: 40	\n import•numpy•as•np\n from•tqdm•import•tqdm\n\n #•digits:•1000•\n l•=•[]\n for•i•in•range(1000):\n ••••l.append(str(i).zfill(3)))\n
Punct 100k Encoded length: 43	\n import•numpy•as•np\n from•tqdm•import•tqdm\n\n #•digits:•1000•\n l•=•[]\n for•i•in•range(1000):\n ••••l.append(str(i).zfill(3)))\n
Merged 80k Encoded length: 44	•\n import•numpy•as•np\n from•tqdm•import•tqdm\n\n #•digits:•1000•\n l•=•[]\n for•i•in•range(1000):\n ••••l.append(str(i).zfill(3)))\n

Table 12. Comparison of tokenizer encoding for a code sample. Spaces in the tokens are indicated with •, token delimitations are indicated by alternating colors.