Data Mixture Inference Attack: BPE Tokenizers Reveal Training Data Compositions

Anonymous Authors[1]

Abstract

The pretraining data of today's strongest language models remains opaque, even when their parameters are open-sourced. In particular, little is known about the proportions of different domains, languages, or code represented in the data. In this work, we tackle a task which we call *data mixture inference*, which aims to uncover the distributional make-up of the pretraining data. We introduce a novel attack based on a previously overlooked source of information — byte-pair encoding (BPE) tokenizers, used by the vast majority of modern language models. Our key insight is that the ordered vocabulary learned by a BPE tokenizer naturally reveals information about the token frequencies in its training data. Given a tokenizer's merge list along with data samples for each category of interest (e.g., the set of natural languages), we formulate a linear program that solves for the relative proportion of each category in the tokenizer's training set. Importantly, to the extent to which tokenizer training data is representative of the pretraining data, we indirectly learn about the pretraining data. In controlled experiments, we show that our attack can recover mixture ratios with high precision for tokenizers trained on known mixtures of natural languages, programming languages, and data sources. We then apply our approach to off-the-shelf tokenizers released alongside recent LMs. We confirm much publicly disclosed information about these models, and also make several new inferences: GPT-4O is much more multilingual than its predecessors, training on 10× more non-English data than GPT-3.5, CLAUDE is trained on predominantly code, and many recent models are trained on 7-16% books.

1. Introduction

Pretraining data is at the heart of language model development, yet it remains a trade secret for today's strongest models. While it has become more common for model-producing organizations to open-source model parameters, they rarely share the pretraining data or many details about its collection. In particular, little is known about the proportion of different languages, code, or data sources present in the data; these design decisions require extensive experimentation that few organizations have the resources to perform, and have a significant impact on the resulting LM (Albalak et al., 2024; MA et al., 2024; Li et al., 2022; Longpre et al., 2023; Schäfer et al., 2024).

While a long line of membership inference attacks (Carlini et al., 2021; Shi et al., 2024; Mireshghallah et al., 2023) aim to reveal information about the model's pretraining data, they typically focus on testing whether particular instances, authors, or websites contributed to the data. In this work, we tackle a different task we call *data mixture inference*, which, given a set of disjoint categories that cover the pretraining data (e.g., the set of natural and programming languages), aims to uncover the proportion of each one.

To this end, we identify a previously overlooked source of information — trained byte-pair encoding tokenizers (BPE; (Sennrich et al., 2016)), which are the near-universal choice for modern language models. Our key insight is that the *ordered merge rules* learned by a BPE tokenizer *naturally reveal information about the frequency of tokens* in the tokenizer's training data. During training, BPE tokenizers iteratively find the ordered pair of tokens with the highest frequency, add it to the merge list, and apply it to the dataset before finding the next highest-frequency pair. Therefore, if the pair (`;`, `\n`) was merged in the 51st step (as in the case of GPT-4O), then it must be the most frequent pair in the data after applying the 50 preceding merges; in this case, it is a signature of substantial code data. Our method builds a linear program where the constraints are derived from the true most-frequent merge at every step in the merge list, and solves for the proportions of each category.

Importantly, the tokenizer training data is ideally representative of the LM's pretraining data (Workshop, 2023); dis-

[1] Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the ICML 2024 Workshop on Foundation Models in the Wild. Do not distribute.

connects lead to poor encoding of the pretraining text (Ahia et al., 2023) and potential for "glitch tokens" that trigger degenerate model behavior (Rumbelow & Watkins, 2023; Geiping et al., 2024; Land & Bartolo, 2024). Note that open-source models require open tokenizers, as the learned merge rules are applied, in order, to tokenize new text. Even for closed models behind APIs, open tokenizers are useful for estimating the cost of queries ahead of time.

We first demonstrate the effectiveness of our method in controlled experiments where we train tokenizers on known mixtures of data. We consider three kinds of data mixtures: natural languages, programming languages, and data sources. Our method is highly effective, achieving between three and six orders of magnitude lower mean squared error than random guessing.

Next, we apply our method to infer previously unknown distributional information from off-the-shelf, commercial tokenizers (the start of these merge lists are shown in §E.1 for qualitative inspection). We consider the tokenizers released with GPT-2, GPT-3.5, GPT-4O, LLAMA 3, and CLAUDE. We corroborate reported information and public intuition about these tokenizers with exact numbers – GPT-2 is trained on predominantly English (98.2%), and GPT-3.5 is the first in the GPT family to be trained extensively on code (62.1%). We also make several new inferences: GPT-4O is trained on 10× more non-English data than GPT-3.5 (39.0% compared to 4.4%), LLAMA 3 and CLAUDE are both trained on $\sim$60% code, and all the models we study are trained on at least 7% book data.

Inferring pretraining data mixtures of LMs has several important implications. It leaks technical information about the model construction, which model producers may intend to keep proprietary. Identifying the exact data sources can potentially enable targeted data poisoning attempts if the LM is further trained on the same sources (Carlini et al., 2024). Finally, data mixture inference can enable external auditing of the pretraining data for biases, such as identifying under-represented languages or data sources.

2. Background: BPE tokenizers

Byte-pair encoding (BPE), introduced by Sennrich et al. (2016) for NLP,[1] is a tokenization algorithm that learns subword-based encodings from training data. Broadly, the algorithm first separates the training corpus into individual bytes (which are used to initialize the vocabulary), and then iteratively merges frequently co-occurring pairs of tokens until the desired vocabulary size is reached.

More precisely, the training text is first *pretokenized*, that is, split into "word" units that give an upper bound on what tokens can look like. Merges cannot bridge these words, and the final learned tokens will be substrings of these words. Pretokenization can be simple as splitting on whitespace, so that common sequences of words (e.g., "*it is*") do not become a single token.

After pretokenization, all of the words are split into bytes, which form the starting vocabulary. Then, the BPE algorithm iteratively counts the frequency of each neighboring pair of tokens and picks the most frequent to be the next merge. This merge is added to the merge list and applied to the entire text, and the merged token is added to the vocabulary. For instance, if the merge is (`th`, `e`), then all instances of "`th e`" will be replaced with `the`, which is added to the vocabulary. BPE then updates the frequencies of all pairs, and identifies the next most frequent. This continues until the desired vocabulary size is reached. At the end of training, the algorithm has learned an ordered list of merge rules $m^{(1)}, ..., m^{(M)}$.

To tokenize new text, the tokenizer splits the text into bytes and applies the learned merge rules, in order. As we will see, this list reflects rich distributional information about the training data.

3. Data mixture inference attack

Suppose we have a set of n data categories of interest, and data distributions $\{\mathcal{D}_i\}_{i=1}^n$ for each one. Then suppose we receive a BPE tokenizer, which was obtained by training on a large sample of text from the mixture $\sum_{i=1}^n \alpha_i^* \mathcal{D}_i$ where the nonnegative weights $\alpha^* \in \mathbb{R}^n$ satisfy $\sum_{i=1}^n \alpha_i^* = 1$. Given corpora $\{D_i\}_{i=1}^n$ sampled from each of the $\mathcal{D}_i$ respectively, the goal of *data mixture inference* is to produce a good estimate $\hat{\alpha}$ of α^*.

In the appendix, we describe how to set up the set of constraints that make up a linear program whose solution is this estimate (§A.1), reduce the storage requirement (§A.2), and improve efficiency (§A.3, §A.4).

4. Experiments

In our initial experiments, we train tokenizers on known data mixtures and measure the accuracy of our attack's prediction. We consider mixtures of natural languages, programming languages, and data sources (which we also refer to as domains). For more detail about each of these datasets, see Appendix D.

4.1. Setup

Because BPE tokenizers operate on bytes, we measure the proportion of each language in terms of bytes. Each tokenizer is trained on a mixture of n categories, where n varies

[1] Though, it originated in 1994 in the field of data compression (Gage, 1994).

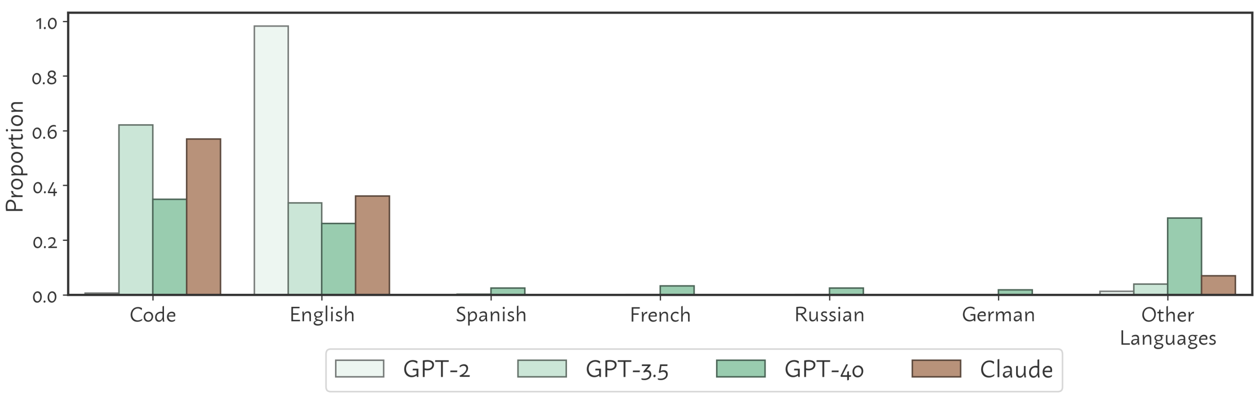


Figure 1. **Our inference for the data mixtures of several commercial tokenizers.**

from 5 and 112. For each value of n, we randomly sample n categories and their weights (using the algorithm from (Smith & Tromble, 2004)) to train 100 tokenizers on 10 GB of data. The data for each category is sampled from the corresponding corpus; if there is not enough data for any category (e.g., we have many low-resource languages), we duplicate the data until the necessary amount is achieved, to preserve the desired mixture ratio. We train tokenizers using the HuggingFace `tokenizers` library and apply a minimal set of common pretokenization operations: we encode strings into bytes using UTF-8, split on whitespace, and only allow digits to be merged with other contiguous digits.

After training the tokenizers, we apply our attack. We estimate merge frequencies for each category by sampling 1 GB of data per category, or less if there is not that much data. Note that the data used for training the tokenizer and estimating pair frequencies are sampled from the same distribution, but are not necessarily the same data. For efficiency, we truncate the merge list to the first $T = 3000$ merges. In Appendix B, we study how our attack scales with the available data and the truncation parameter T.

We report the mean squared error of the estimated proportions, $\text{MSE} := \frac{1}{n}\sum_{i=1}^{n}(\hat{\alpha}_i - \alpha_i^*)^2$. In practice, we use $\log_{10}(\text{MSE})$ for readability. We empirically calculate the accuracy of random guessing and report it as a baseline.

Natural Language Mixtures We use the Oscar v23.01 corpus (Abadji et al., 2022), which is based on the Nov/Dec 2022 dump from Common Crawl. We consider the 112 languages with at least 1MB of data.

Programming Language Mixtures We use the GitHub split of RedPajama (Computer, 2023), an open-source reproduction of the LLAMA training data, with a total of 37 programming languages.

Table 1. **Experimental results for controlled experiments.** The settings we consider are mixtures of natural languages, mixtures of programming languages, and mixtures of data sources. In each cell, we report the mean and standard deviation of $\log_{10}(\text{MSE})$ over 100 trials. Note that a decrease by 1 corresponds to a 10× improvement in the MSE. **Rand** shows the random-guessing baseline. In total, we have 112 natural languages, 37 programming languages, and 5 domains.

n	Rand	**Languages**	**Code**	**Data Sources**
5	-1.39	$-6.61_{\pm 2.10}$	$-6.51_{\pm 0.79}$	$-3.90_{\pm 0.94}$
10	-1.84	$-7.65_{\pm 0.66}$	$-6.30_{\pm 0.67}$	–
30	-2.70	$-7.58_{\pm 1.29}$	$-5.95_{\pm 1.18}$	–
112	-3.82	$-7.77_{\pm 0.90}$	–	–

Domain Mixtures We consider the following five English domains, using data from RedPajama: **Wikipedia**, containing English Wikipedia dumps from Jun-Aug 2022, **Web**, Common Crawl data that was de-duplicated and filtered for English, **Books** from the Gutenberg Project and Books3 of The Pile, **Code** from GitHub, and **Academic**, which contains latex files of scientific papers on ArXiv.

4.2. Results

Shown in Table 1, our attack is highly effective across the board. Across all mixture types and values of n, we achieve mean MSE 3 to 5 orders of magnitude better than random guessing. In general, we observe that the performance decreases slightly from mixed languages to mixed code to mixed domains. This is perhaps unsurprising when considering the source of signal for our attack, which is the different token pair frequencies in different categories of data. Intuitively, we would expect these to be very distinct for different natural languages, which have different vocabularies. In contrast, programming languages can share many syntactic features, such as using indents, semicolons, curly brackets `{}`, and control structures like `if`, `for`, and while. Likewise, different English domains should largely share the same vocabulary, but have more subtle differences in token frequencies. Nonetheless, even in this most challenging setting, we achieve accuracy 1000× better than random.

5. Attacking commercial tokenizers

After validating our attack in synthetic experiments (§4), we apply it to infer training data mixtures of off-the-shelf commercial tokenizers. We refer to tokenizers by the name of the model they were first released with, whose pretraining data they most likely reflect. We consider LLAMA 3, GPT-2, GPT-3.5, GPT-4O, and CLAUDE. While the last three are closed models, they publicly share their tokenizers so that customers can estimate the cost of queries ahead of time.

For these experiments, we aim to infer the proportion of natural languages and code (where code is a single category, not split into separate programming languages), as well as the proportion of English domains that make up the English data. We merge code into one category because some programming languages, like Markdown and Pod6, are almost entirely English, and we do not expect the distribution of programming languages in pretraining data to differ substantially from that of GitHub, which is the largest public code hosting platform. To infer the distribution of English domains, we replace the English category with the four English domains from §4 (web, books, Wikipedia, and academic). We expect these to approximately cover the English data.

Our predictions are shown in Figure 1, with specific numbers in §D.3. Below, we discuss our findings in comparison with publicly disclosed information about these models.

5.1. GPT models

The GPT tokenizers are open-source on tiktoken. There are three such tokenizers, released with GPT-2, GPT-3.5, and the very recent GPT-4O.

GPT-2 GPT-2 (Radford et al., 2019) was trained on WebText, consisting of text scraped from outbound links from Reddit, and filtered to be English-only. Indeed, we confirm the training data consists of 98.2% English. However, we surprisingly estimate that only 57.5% of the data was web, with another 39.4% being books — even though books were not explicitly included in WebText. In a data contamination analysis, the authors indeed report that they find books in WebText, but our estimate suggests the contamination is much deeper. We note that books were a popular source of pretraining data for early Transformer language models, with GPT-1 being trained entirely on BooksCorpus (Zhu et al., 2015). The GPT-2 tokenizer was reused for GPT-3.

GPT-3.5 The GPT-3.5 family of models is known to depart from its predecessors by training on large amounts of code: the first two models in this family were code-davinci-002 (trained on text and code) and text-davinci-002, with the latter being instruction-tuned from the former. In fact, some evidence suggests that GPT-3.5's large leap in reasoning abilities comes from this code data, which intuitively requires similar procedural skills (Fu & Khot, 2022).

Indeed, we estimate that GPT-3.5 is trained on 62.1% code, compared to $< 1\%$ for GPT-2. In the domain breakdown, we see that 25.2% is of the data is web, 7.7% books, and 0.6% academic articles. The substantial representation of books (though much lower than GPT-2) is consistent with findings that this model has memorized a wide collection of copyrighted books (Chang et al., 2023).

GPT-4O Released on May 13, 2024, GPT-4O (OpenAI, 2024) is a multimodal model announced as more multilingual than its predecessors; its tokenizer achieves a better compression rate on non-English languages, and the model has notably better non-English performance.

Our findings support this. We estimate that GPT-4O was trained on 39.0% non-English data, compared to only 4.3% for GPT-3.5, making it the most multilingual model we study. The language distribution has a thick non-English tail, with 58 languages that make up at least 0.1% of the data: the most common are French (3.2%), Spanish (2.5%), Russian (2.4%), Portuguese (2.4%), and German (1.8%). Additionally, GPT-4O was trained on 9.4% books.

5.2. CLAUDE

Finally, we consider the CLAUDE tokenizer. Very little is known about tokenizers from the CLAUDE family, but a remark in the Anthropic SDK suggests that Claude 1 (Anthropic, 2023a) and 2 (Anthropic, 2023b) share the same tokenizer, which is open-source, while Claude 3 (Anthropic, 2024) uses a different (closed) tokenizer. Nothing is shared about the pretraining data of these models. We estimate that CLAUDE was trained on 57.0% code, 36.1% English, and 6.9% other languages. Moreover, more than half of its English data comes from books.

6. Conclusion

In this work, we present a *data mixture inference* attack that solves for the distributional make-up of a tokenizer's training data, which is commonly representative of the language model's pretraining data. This can shed light on the secretive and often contentious design decisions surrounding pretraining data today, potentially enabling external auditing for safety, copyright issues, and distributional biases.

References

Abadji, J., Ortiz Suarez, P., Romary, L., and Sagot, B. Towards a cleaner document-oriented multilingual crawled corpus. In Calzolari, N., Béchet, F., Blache, P., Choukri, K., Cieri, C., Declerck, T., Goggi, S., Isahara, H., Maegaard, B., Mariani, J., Mazo, H., Odijk, J., and Piperidis, S. (eds.), *Proceedings of the Thirteenth Language Resources and Evaluation Conference*, pp. 4344–4355, Marseille, France, June 2022. European Language Resources Association. URL https://aclanthology.org/2022.lrec-1.463.

Ahia, O., Kumar, S., Gonen, H., Kasai, J., Mortensen, D., Smith, N., and Tsvetkov, Y. Do all languages cost the same? tokenization in the era of commercial language models. In Bouamor, H., Pino, J., and Bali, K. (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 9904–9923, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.614. URL https://aclanthology.org/2023.emnlp-main.614.

Albalak, A., Elazar, Y., Xie, S. M., Longpre, S., Lambert, N., Wang, X., Muennighoff, N., Hou, B., Pan, L., Jeong, H., Raffel, C., Chang, S., Hashimoto, T., and Wang, W. Y. A survey on data selection for language models, 2024. URL https://arxiv.org/abs/2402.16827.

Anthropic. Introducing claude, 2023a. URL https://www.anthropic.com/news/introducing-claude.

Anthropic. Claude 2, 2023b. URL https://www.anthropic.com/news/claude-2.

Anthropic. Introducing the next generation of claude, 2024. URL https://www.anthropic.com/news/claude-3-family.

Benders, J. Partitioning procedures for solving mixed-variables programming problems. *Numerische mathematik*, 4(1):238–252, 1962.

Carlini, N., Tramèr, F., Wallace, E., Jagielski, M., Herbert-Voss, A., Lee, K., Roberts, A., Brown, T., Song, D., Erlingsson, Ú., Oprea, A., and Raffel, C. Extracting training data from large language models. In *30th USENIX Security Symposium (USENIX Security 21)*, pp. 2633–2650. USENIX Association, August 2021. ISBN 978-1-939133-24-3. URL https://www.usenix.org/conference/usenixsecurity21/presentation/carlini-extracting.

Carlini, N., Jagielski, M., Choquette-Choo, C., Paleka, D., Pearce, W., Anderson, H., Terzis, A., Thomas, K., and Tramèr, F. Poisoning web-scale training datasets is practical. In *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 179–179, Los Alamitos, CA, USA, may 2024. IEEE Computer Society. doi: 10.1109/SP54263.2024.00179. URL https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00179.

Chang, K., Cramer, M., Soni, S., and Bamman, D. Speak, memory: An archaeology of books known to ChatGPT/GPT-4. In Bouamor, H., Pino, J., and Bali, K. (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 7312–7327, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.453. URL https://aclanthology.org/2023.emnlp-main.453.

Computer, T. Redpajama: An open source recipe to reproduce llama training dataset, 2023. URL https://github.com/togethercomputer/RedPajama-Data.

Dantzig, G. B. and Wolfe, P. Decomposition principle for linear programs. *Operations Research*, 8(1):101–111, 1960. URL https://www.jstor.org/stable/167547.

Fu, Yao; Peng, H. and Khot, T. How does gpt obtain its ability? tracing emergent abilities of language models to their sources. *Yao Fu's Notion*, Dec 2022. URL https://yaofu.notion.site/How-does-GPT-Obtain-its-Ability-Tracing-Emergent-

Gage, P. A new algorithm for data compression. *The C Users Journal archive*, 12:23–38, 1994. URL https://api.semanticscholar.org/CorpusID:59804030.

Geiping, J., Stein, A., Shu, M., Saifullah, K., Wen, Y., and Goldstein, T. Coercing llms to do and reveal (almost) anything, 2024. URL https://arxiv.org/abs/2402.14020.

Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023. URL https://www.gurobi.com.

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: System Demonstrations*, pp. 66–71, Brussels, Belgium, November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-2012. URL https://aclanthology.org/D18-2012.

Land, S. and Bartolo, M. Fishing for magikarp: Automatically detecting under-trained tokens in large language models, 2024. URL https://arxiv.org/abs/2405.05417.

Li, M., Gururangan, S., Dettmers, T., Lewis, M., Althoff, T., Smith, N. A., and Zettlemoyer, L. Branch-train-merge: Embarrassingly parallel training of expert language models, 2022. URL https://arxiv.org/abs/2208.03306.

Limisiewicz, T., Blevins, T., Gonen, H., Ahia, O., and Zettlemoyer, L. Myte: Morphology-driven byte encoding for better and fairer multilingual language modeling, 2024. URL https://arxiv.org/abs/2403.10691.

Longpre, S., Yauney, G., Reif, E., Lee, K., Roberts, A., Zoph, B., Zhou, D., Wei, J., Robinson, K., Mimno, D., and Ippolito, D. A pretrainer's guide to training data: Measuring the effects of data age, domain coverage, quality, & toxicity. 2023. URL https://arxiv.org/abs/2305.13169.

MA, Y., Liu, Y., Yu, Y., Zhang, Y., Jiang, Y., Wang, C., and Li, S. At which training stage does code data help LLMs reasoning? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=KIPJKST4gw.

Mireshghallah, N., Vogler, N., He, J., Florez, O., El-Kishky, A., and Berg-Kirkpatrick, T. Simple temporal adaptation to changing label sets: Hashtag prediction via dense KNN. In Bouamor, H., Pino, J., and Bali, K. (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 7302–7311, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.452. URL https://aclanthology.org/2023.emnlp-main.452.

OpenAI. Hello GPT-4o, 2024. URL https://openai.com/index/hello-gpt-4o/.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. 2019. URL https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

Rumbelow, J. and Watkins, M. Solidgoldmagikarp (plus, prompt generation), 2023. URL https://www.lesswrong.com/posts/aPeJE8bSo6rAFoLqg/solidgoldmagikarp-plus-prompt-generation.

Schäfer, A., Ravfogel, S., Hofmann, T., Pimentel, T., and Schlag, I. Language imbalance can boost cross-lingual generalisation, 2024. URL https://arxiv.org/abs/2404.07982.

Sennrich, R., Haddow, B., and Birch, A. Neural machine translation of rare words with subword units. In Erk, K. and Smith, N. A. (eds.), *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.

Shi, W., Ajith, A., Xia, M., Huang, Y., Liu, D., Blevins, T., Chen, D., and Zettlemoyer, L. Detecting pretraining data from large language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=zWqr3MQuNs.

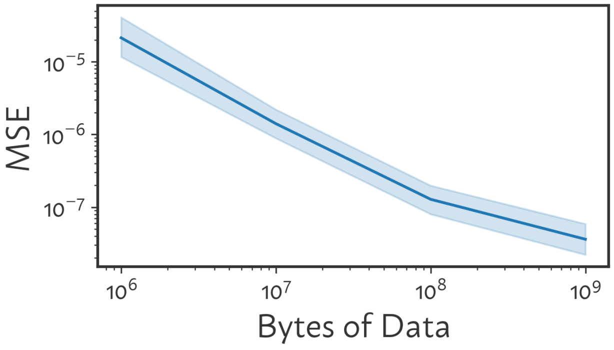
Smith, N. A. and Tromble, R. Sampling uniformly from the unit simplex. Technical report, 2004. URL https://www.cs.cmu.edu/~nasmith/papers/smith+tromble.tr04.pdf.

Wang, J., Gangavarapu, T., Yan, J. N., and Rush, A. M. Mambabyte: Token-free selective state space model, 2024. URL https://arxiv.org/abs/2401.13660.

Workshop, B. Bloom: A 176b-parameter open-access multilingual language model, 2023. URL https://arxiv.org/abs/2211.05100.

Yang, J. Rethinking tokenization: Crafting better tokenizers for large language models, 2024. URL https://arxiv.org/abs/2403.00417.

Zhu, Y., Kiros, R., Zemel, R., Salakhutdinov, R., Urtasun, R., Torralba, A., and Fidler, S. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.



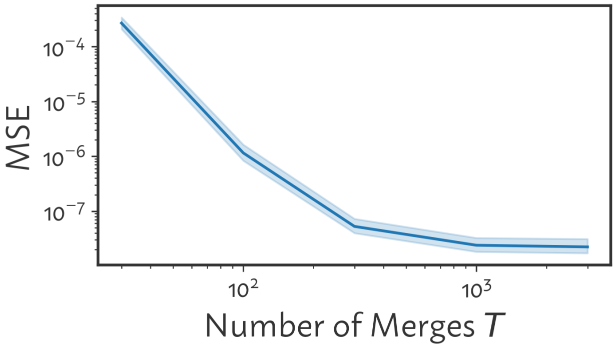


Figure 2. **Analysis of scaling with the amount of data used for estimating pair frequencies** (§B.1). Sampling more data per category consistently produces more precise inferences.

Figure 3. **Analysis of scaling with the number of merges** T used in the merge list (§B.2). Attack performance saturates at around 1000 merges.

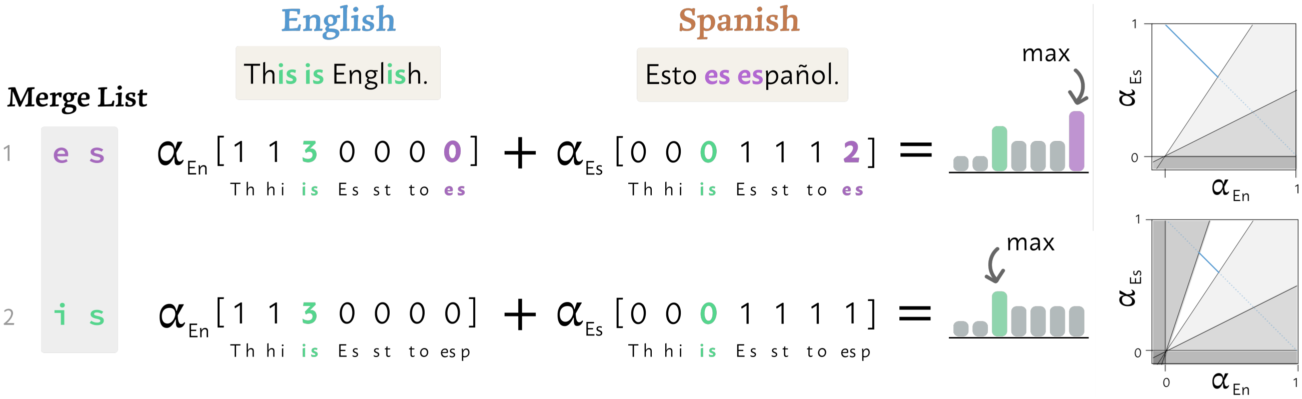


Figure 4. **Illustration of our method** on a simple example. We know that after applying in the first $t-1$ merges to the training data, the t^{th} merge be the most common pair. More explicitly, this means that α_i should give a vector in which the value corresponding to the true next merge is the maximum. Our attack collects these inequalities at every time step to construct the linear program.[3]

A. Implementation

In this section, we describe the implementation of our algorithm, introduced in §3.

A.1. Data mixture inference via linear programming

We build a linear program (LP) with variables α and constraints derived using information from the tokenizer and our sample corpora. The given tokenizer can be represented by an ordered list of merge rules $m^{(1)}, \ldots, m^{(M)}$. For each time step $t \in [M]$, we apply all preceding merge rules $m^{(1)}, \ldots, m^{(t-1)}$ to our corpora D_i and use $c^{(t)}_{i,p}$ to denote how many times the token pair p occurred in the partially merged text. We know that when the tokenizer was trained, the pair $m^{(t)}$ was more frequent at time t than any other pair. In other words,

$$\sum_{i=1}^{n} \alpha_i c^{(t)}_{i,m^{(t)}} \geq \sum_{i=1}^{n} \alpha_i c^{(t)}_{i,p} \qquad \text{for all } p \neq m^{(t)}.$$

Collecting these constraints for all t and p defines a set of possible α's.

Of course, because we do not have access to the exact data the tokenizer was trained on, but only samples from the same component distributions, the linear program we described above may not be feasible, as there will be some variation in the

[3] Technically, the elements of the vectors should be normalized by size of the language data, but in this case they are the same for the two languages so we show the unnormalized counts for readability.

counts due to sampling. To address this, we relax the constraints by introducing new non-negative variables $v^{(t)}$ for $t \in [M]$, and v_p for all pairs p, which represent the degree of constraint violation. *[can we explain why there are two new variables introduced here? why do we provide slack for merges and for candidate pairs, separately?]*$_{-AL}$ We replace our constraints with new ones of the form

$$v^{(t)} + v_p + \sum_{i=1}^{n} \alpha_i c_{i,m^{(t)}}^{(t)} \geq \sum_{i=1}^{n} \alpha_i c_{i,p}^{(t)} \qquad \text{for all } p \neq m^{(t)}.$$

This new system of constraints is guaranteed to be feasible as the v's can be made arbitrarily large. In general, we expect $v^{(t)}$ to be large when $m^{(t)}$ is over-represented in the tokenizer training data and v_p to be large when p is over-represented in the mixture defined by α. To produce the best possible estimate, our objective is to minimize the total constraint violation $\sum_{t=1}^{M} v^{(t)} + \sum_p v_p$. We call the resulting linear program `LP1`. To estimate α, we solve `LP1` and report the optimal value of α as our $\hat{\alpha}$.

As written, `LP1` can be prohibitively large. If our vocabulary has size V, the total number of constraints scales like $O(V^3)$ since there are $O(V)$ time steps t to consider and $O(V^2)$ competing byte pairs $p \neq m^{(t)}$. Additionally, there are $O(V^2)$ variables v_p. The first step to reduce the size is to limit t to the first T merges. We will call this truncated program $\texttt{LP1}_T$. However, even for modest choices of T, $\texttt{LP1}_T$ can still have millions of variables and tens of billions of constraints. In the following sections, we will describe how to efficiently solve $\texttt{LP1}_T$ using simultaneous delayed row (constraint) and column (variable) generation (Benders, 1962; Dantzig & Wolfe, 1960).

A.2. Efficient storage of pair counts

First, as a preprocessing step, we apply the target tokenizer to each language corpus D_i, recording the pair counts $c_{i,p}^{(t)}$ after each merge is applied for later use. Naively, this would require a large amount of space, since the number of possible pairs p scales like $O(V^2)$. However, note that $c_{i,p}^{(t)} \neq c_{i,p}^{(t+1)}$ only when p overlaps with $m^{(t)}$. In other words, bigrams with no overlap with the most recent merge will have their counts unaffected. Thus, there are only $O(V)$ differences between $c_{i,\cdot}^{(t)}$ and $c_{i,\cdot}^{(t+1)}$. In practice, the number of changes caused by a single merge is usually a few hundred at most. By saving only the incremental changes from each set of pair counts to the next, we can efficiently record the pair counts at every iteration of the tokenization process.

A.3. Efficient constraint violation detection

Our plan is to solve $\texttt{LP1}_T$ using only a subset of its constraints, giving a potential solution (α, v). We can check whether (α, v) violates any of the constraints of $\texttt{LP1}_T$ and if it does, we can add them to the subset. This requires an efficient method to detect violated constraints, which we describe below.

For convenience, let $s_p^{(t)} := \sum_{i=1}^{n} \alpha_i c_{i,p}^{(t)}$ and recall that, for a given time step t, we want to check whether $v^{(t)} + s_{m^{(t)}}^{(t)} \geq \max_p (s_p^{(t)} - v_p)$. Considered in isolation, the only thing we can do is scan over all possible $p \neq m^{(t)}$ to see if the constraint is violated, which can be quite costly. Moreover, we must do this for many different t. However, by taking advantage of the structure of the $s_p^{(t)}$ as t varies, we can reduce our work substantially.

The first step is to take each initial pair p and add it to a priority queue with priority $s_p^{(0)}$. This can be done in aggregate in $O(V^2)$ time using a fast heap building algorithm. Now, we can scan through the pairs in order of descending $s_p^{(0)}$ using the queue's `delete-min` operation. For each pair p, we can check whether $v^{(t)} + s_{m^{(t)}}^{(t)} > s_{p^*}^{(t)} - v_{p^*}$ and if not, we mark the corresponding constraint as violated. Once we find $p = m^{(t)}$, we stop, since the remaining constraints must be satisfied. If there were k pairs before $m^{(t)}$ in the queue, then the total time taken is $O(k \log V)$.

Crucially, we can quickly update the priority queue to reflect the state at $t = 1$. Since we precomputed all count changes from $c_{i,p}^{(0)}$ to $c_{i,p}^{(1)}$, we know what queue entries need to be updated or inserted. If k_{new} new pairs were created and k_{old} pairs had their counts changed when pair $m^{(0)}$ was merged, then we can update the priority queue using k_{new} `insert` operations and k_{old} `decrease-priority` operations, which can be done in $O((k_{\text{new}} + k_{\text{old}}) \log V)$ time. Now that we have updated the priority queue, it is easy to check for any constraint violations for $t = 1$. By iterating this process, we can quickly check for violated constraints for all t.

A.4. Lazy constraint generation

Now we are ready to solve $\texttt{LP1}_T$ in an efficient way. We begin by guessing uniform proportions for α, and $v^{(t)} = v_p = 0$ for all t, p. Then we use our constraint checker to identify violated constraints of $\texttt{LP1}_T$ and construct a lazy version of $\texttt{LP1}_T$, denoted $\texttt{LP2}_T$, using only those constraints and the variables they contain. We then solve $\texttt{LP2}_T$, which gives us a new guess for α. We iterate the above process, adding progressively more constraints to $\texttt{LP2}_T$ until we find a solution that is also feasible for $\texttt{LP1}_T$. It follows that this solution is optimal for $\texttt{LP1}_T$ since the two programs share the same objective. This is guaranteed to happen eventually because there are a finite number of constraints to add.

In practice, with our choice of $T = 3000$ and the datasets we use, the constraint violation detection typically takes under a minute and 50 iterations is always sufficient to converge on a solution. Thus our total runtimes are generally under an hour, with easy instances completing much faster.

B. Scaling analysis

Using the natural language mixture setup from §4 with $n = 10$ languages, we analyze how our attack's performance varies with the amount of data used (§B.1) and the number of merges we apply from the merge list (§B.2).

B.1. How many data samples do we need from each category?

[*I would like for us to have 100 trials for this experiment*]$_{-AL}$ We explore how the attack's performance scales with the amount of data sampled from each distribution $\mathcal{D}_i$ for calculating pair counts. We train 10 new tokenizers considering only languages from our corpus with >1 GB of data. Then for our attack, we compare sampling 1 MB, 10 MB, 100 MB, and 1 GB of data for each language. Shown in Figure 2, more data seems to consistently improve the accuracy of predictions. [*maybe need to mention that we fix T?*]$_{-AL}$

B.2. How many merges do we need to consider?

Next, we investigate how performance scales with the number of merges T that we apply from the merge list. We reuse the 100 tokenizers from §4, and solve for the data mixture using various choices of $T \in [30, 3000]$ for the total number of merges considered. Shown in Figure 3, we find that performance saturates at about 1000 merges. [*need to mention that we fix the amount of data used*]$_{-AL}$

C. Possible defenses

We discuss some possible approaches for defenses to our attack, which we believe would all have limited effectiveness.

Post-hoc changing the order of merge rules Model producers may consider changing the order of merge rules, which is the source of signal for our attack, after the tokenizer is trained. However, naively re-ordering merge rules for a tokenizer would be damaging, as it can lead to unfamiliar encodings of words as well as entirely unreachable tokens. The only functionally-equivalent re-ordering would be *within contiguous sections* of merge rules that have *no* overlap in used bytes. In this case, we can easily adapt our method by working at the level of contiguous non-conflicting sections instead of individual merge rules, as we know that each section has higher frequencies than the next. Moreover, we expect these sections to be quite small.

Hiding pretokenization rules Our method relies on a reasonable reconstruction of the pretokenization rules, which control what kinds of merge rules are considered. It is not necessary to share these pretokenization rules, as they are not strictly necessary for inference. However, we find that important pretokenization rules (like whether to pre-tokenize on spaces, digits, and punctuation) are easy to reconstruct from manual inspection. Moreover, organizations are incentivized to release pretokenization rules, as it greatly enhances the speed of encoding by enabling parallelism.

Not using BPE tokenizers Model producers may choose to forgo BPE tokenizers entirely in future models. Despite the current popularity of BPE, there is a lively area of research into alternative methods of tokenization or doing without tokenization entirely (Wang et al., 2024; Yang, 2024; Limisiewicz et al., 2024). While we only explore BPE tokenizers in this paper, it is plausible that *any* tokenizer learning algorithm will leak information about its training data, as they are specifically developed to best encode the given data.

D. Experiment details

D.1. Data details

The full set of categories that we use in §4 and the amount of data available for each category are shown in Table 4, Table 5, and Table 6.

Below, we enumerate the licenses for each dataset we use.

- Oscar: CC0 1.0 Universal
- RedPajama has different licenses for each subset
  - C4: ODC-BY
  - GitHub: MIT, BSD, or Apache
  - Books3: MIT
  - Project Gutenberg: Apache 2.0
  - ArXiv: CCo 1.0
  - Wikipedia: CC-BY-SA-3.0

D.2. Compute details

We run all of our experiments on CPUs. For training tokenizers and calculating pair frequencies, we use 16 CPUs and a variable amount of memory (ranging from 4 GB to 64 GB) depending on the data. Training a tokenizer on 10 GB of data (as in our experiments) usually takes around 10 minutes, while calculating pair counts takes between 1 minute and 2 hours, again depending on the data. To solve our linear programs, we use Gurobi (Gurobi Optimization, LLC, 2023).

D.3. Full results

We report the full inferences in Table 3.

E. Commercial tokenizers

E.1. Snapshot of commercial tokenizer merge lists

We show the first 50 merges of the commercial tokenizers we study in Table 2

E.2. Handling redundant merges

We observe that the merge list of Llama 3 contains clusters of redundant merge rules. For instance, we see the sequence of merges _ `the`, _`t he`, and _`th e`, as well as _ `and`, _`a nd`, and _`an d`. Because the merge path for every token is unique, it is impossible for more than one of these merges to ever be used, and we empirically verify this by applying the tokenizer to a large amount of text. This phenomenon is not unique to Llama 3 — we also find it for Gemma and Llama tokenizers.

We find that this is an artifact of the conversion from `sentencepiece` to Huggingface `tokenizers` format. To construct the merge list, the conversion algorithm naively combines every pair of tokens in the vocabulary, and then sorts them by token ID, which represents order of creation. While this is functionally correct, because the redundant merges are not products of the BPE algorithm (i.e., they do not actually represent the most-likely next-merge), we need to remove them for our algorithm. To do this, we do some simple pre-processing: for every cluster of redundant merges, we record the path of merges that achieves each merge; the earliest path is the one that would be taken, so we keep that merge and remove the rest.

As an aside, note that this means the merge list can be completely reconstructed from the vocabulary list in the order of token creation.

E.3. GPT tokenizers

While GPT tokenizers are open source on `tiktoken`, they are not released in a format compatible with HuggingFace tokenizers. We used the `tokenizers`-compatible files uploaded by a HuggingFace user named `Xenova`. For instance, the GPT-4O tokenizer can be found at https://huggingface.co/Xenova/gpt-4o.

E.4. Miscellaneous observations

LLAMA and GEMMA tokenizers were trained using `sentencepiece` (Kudo & Richardson, 2018).

While merges of spaces are allowed by LLAMA, all of these merges occur at the very end of the merge list, suggesting that they were manually added.

Table 2. **The first 50 merges of the commercial tokenizers we study.** For readability, we replace the space token Ġ with _ and the newline Ċ with \n. To a careful observer, there are many interpretable signatures of training sources. For instance, consecutive whitespace is common in code, as indents are equivalent to four spaces by most coding standards. Indeed, `_ _` is the first merge of all tokenizers except GPT-2, which is likely the only tokenizer not trained on code. The merges `; \n`, `_ =`, and `sel f` are also common token pairs in code. The odd-looking symbols are encodings of bytes that make up *parts* of single characters in many languages. For instance, `à ¤` encodes the prefix for the first half of the Devanagari Unicode block (used for writing Hindi, Nepali, Sanskrit, among others), `Ð °` encodes the Cyrillic "a", and `á ĥ` encodes the prefix for the second half of the Georgian Unicode block. The early presence of these in GPT-4o is a sign of its multilingual training data.

GPT-2	GPT-3.5	GPT-4o	Llama 3	Claude
_ t	_ _	_ _	_ _	_ _
_ a	__ __	__ __	__ __	__ __
h e	i n	i n	i n	i n
i n	_ t	e r	_ t	__ _
r e	____ ____	_ t	____ ____	_ t
o n	e r	_ a	e r	e r
_t he	__ _	e n	__ _	____ ____
e r	o n	o n	o n	o n
_ s	_ a	r e	_ a	_ a
a t	r e	_ s	r e	r e
_ w	a t	a t	a t	a t
_ o	s t	o r	s t	s e
e n	e n	e s	e n	h e
_ c	o r	____ ____	o r	o r
i t	_t h	a n	_t h	s t
i s	\n \n	__ _	\n \n	e n
a n	_ c	_ d	_ c	____ ___
o r	l e	h e	l e	a l
e s	_ s	_ c	_ s	_t he
_ b	i t	_ p	i t	i t
e d	a n	i s	a n	_ c
_ f	a r	a r	a r	a n
in g	a l	i t	a l	l e
_ p	_th e	\n \n	_th e	_ =
o u	; \n	a l	; \n	d e
_a n	_ p	à ¤	_ p	a r
a l	_ f	l e	_ f	\n _______
a r	o u	o u	o u	_ f
_t o	_ =	_ m	_ =	_ p
_ m	i s	_ f	i s	\n ________
_o f	____ ___	_ w	____ ___	_ o
_ in	in g	_ b	in g	_ s
_ d	e s	a s	e s	_ w
_ h	_ w	in g	_ w	m e
_an d	i on	_t he	i on	\n ___
i c	e d	i c	e d	r o
a s	i c	e t	i c	i on
l e	_ b	_ o	_ b	in g
_t h	_ d	i on	_ d	i s
i on	e t	e d	e t	_ in
o m	_ m	e l	_ m	_ b
l l	_ o	_ n	_ o	i c
en t	ĉ ĉ	r o	ĉ ĉ	se l
_ n	r o	en t	r o	o u
_ l	a s	_ Ð	a s	sel f
s t	e l	n d	e l	e d
_ re	c t	s t	c t	- -
v e	n d	á ĥ	n d	n d
_ e	_ in	Ð °	_ in	e s
r o	_ h	_ l	_ h	_ m

Table 3. **The full set of inferences** we make for commercial tokenizers. Note that instead of one English category, we split it into four English domains: web, books, academic, and Wikipedia. "Other languages" combines the languages with $<$10 MB of data (see Table 4).

Category	GPT-2	GPT-3.5	GPT-4O	LLAMA 3	CLAUDE
Code	0.6	62.1	34.9	61.9	57.0
Web	57.5	25.2	16.7	22.9	11.0
Books	39.4	7.7	9.4	10.0	16.0
French	0.0	0.1	3.2	0.3	0.0
Spanish	0.0	0.3	2.5	0.5	0.0
Russian	0.0	0.0	2.4	0.1	0.0
Portuguese	0.0	0.4	2.4	0.4	0.9
German	0.0	0.0	1.8	0.0	0.0
Dutch	0.0	0.0	1.7	0.0	0.2
Arabic	0.0	0.0	1.5	0.0	0.0
Hindi	0.0	0.0	1.4	0.0	0.0
Other languages	0.0	0.0	0.9	0.0	0.0
Italian	0.0	0.6	0.8	0.5	0.2
Chinese	0.2	0.3	0.8	0.3	0.5
Polish	0.0	0.3	0.8	0.4	0.1
Turkish	0.1	0.0	0.8	0.0	0.0
Finnish	0.0	0.0	0.8	0.0	0.0
Swedish	0.1	0.0	0.7	0.1	0.0
Georgian	0.0	0.0	0.7	0.0	0.0
Catalan	0.0	0.1	0.7	0.0	0.6
Vietnamese	0.0	0.1	0.7	0.1	0.0
Korean	0.1	0.2	0.6	0.2	0.3
Slovenian	0.0	0.2	0.6	0.2	0.2
Estonian	0.0	0.1	0.6	0.1	0.2
Greek	0.1	0.1	0.5	0.1	0.1
Danish	0.0	0.0	0.5	0.0	0.7
Gujarati	0.0	0.0	0.5	0.0	0.0
Hebrew	0.0	0.0	0.5	0.0	0.0
Malayalam	0.0	0.0	0.5	0.0	0.0
Romanian	0.0	0.4	0.5	0.4	0.4
Bangla	0.0	0.0	0.5	0.0	0.0
Ukrainian	0.0	0.0	0.5	0.0	0.0
Armenian	0.0	0.1	0.4	0.1	0.1
Slovak	0.0	0.0	0.4	0.0	0.0
Czech	0.0	0.0	0.4	0.0	0.0
Persian	0.0	0.0	0.4	0.0	0.0
Japanese	0.1	0.0	0.4	0.0	0.0
Kannada	0.0	0.0	0.3	0.0	0.0
Thai	0.0	0.0	0.3	0.0	0.0
Telugu	0.0	0.0	0.3	0.0	0.0
Icelandic	0.0	0.0	0.3	0.1	0.0
Tatar	0.0	0.0	0.3	0.0	0.0
Tajik	0.0	0.0	0.3	0.0	0.0
Urdu	0.0	0.0	0.3	0.0	0.0
Indonesian	0.0	0.0	0.3	0.0	0.0
Bulgarian	0.0	0.0	0.3	0.0	0.0
Tamil	0.0	0.0	0.3	0.0	0.0
Lithuanian	0.0	0.0	0.3	0.0	0.2
Filipino	0.2	0.1	0.2	0.1	0.2
Bashkir	0.0	0.0	0.2	0.0	0.0
Welsh	0.0	0.3	0.2	0.4	0.2
Hungarian	0.0	0.0	0.2	0.0	0.0
Belarusian	0.0	0.0	0.2	0.0	0.0
Kazakh	0.0	0.0	0.2	0.0	0.0
Pashto	0.0	0.0	0.2	0.0	0.0
Yiddish	0.0	0.0	0.2	0.0	0.0
Sindhi	0.0	0.0	0.1	0.0	0.0
Assamese	0.0	0.0	0.1	0.0	0.0
Azerbaijani	0.0	0.0	0.1	0.0	0.0
Latvian	0.0	0.1	0.1	0.1	0.1
Nepali	0.0	0.0	0.1	0.0	0.0
Serbian	0.0	0.0	0.1	0.0	0.0
Kurdish	0.0	0.0	0.1	0.0	0.0
Kyrgyz	0.0	0.0	0.1	0.0	0.0
Albanian	0.0	0.0	0.1	0.0	0.0
Galician	0.0	0.2	0.1	0.0	0.4
Sinhala	0.0	0.0	0.1	0.0	0.0
Punjabi	0.0	0.0	0.1	0.0	0.0
Divehi	0.0	0.1	0.1	0.1	0.1
Uyghur	0.0	0.0	0.1	0.0	0.0
Burmese	0.0	0.0	0.1	0.0	0.0
Mongolian	0.0	0.0	0.1	0.0	0.0
Khmer	0.0	0.0	0.1	0.0	0.0
Amharic	0.0	0.0	0.0	0.0	0.0
Central Kurdish	0.0	0.0	0.0	0.0	0.0
Odia	0.0	0.0	0.0	0.0	0.0
Tibetan	0.0	0.0	0.0	0.0	0.0
Lao	0.0	0.0	0.0	0.0	0.0
Basque	0.0	0.0	0.0	0.0	0.1
Macedonian	0.0	0.0	0.0	0.0	0.0
Academic	1.3	0.6	0.0	0.4	5.9
Western Panjabi	0.0	0.0	0.0	0.0	0.0
Esperanto	0.0	0.0	0.0	0.0	0.3
Marathi	0.0	0.0	0.0	0.0	0.0
Wiki	0.0	0.0	0.0	0.0	3.1
Norwegian	0.1	0.0	0.0	0.0	0.2

Table 4. The 112 natural languages considered in §4. The data is from Oscar v23.01, which performs language identification at the document level.

Language	Size (MB)	Language	Size (MB)	Language	Size (MB)	Language	Size (MB)
Chinese	776494.9	Bangla	19055.4	Icelandic	2194.7	Sanskrit	56.3
English	666955.4	Hebrew	17970.6	Slovenian	1398.1	Ossetic	50.7
Russian	531902.4	Tamil	15776.8	Punjabi	1377.2	Chuvash	42.3
Spanish	424143.2	Catalan	15346.5	Basque	1195.9	Cebuano	41.1
French	371967.1	Danish	14843.6	Tajik	1028.4	Afrikaans	37.2
German	356683.7	Lithuanian	14518.6	Tatar	834.1	Breton	31.4
Italian	214768.2	Georgian	8388.2	Central Kurdish	773.1	South Azerbaijani	28.4
Japanese	181299.8	Estonian	8026.9	Filipino	719.4	Croatian	26.5
Hungarian	150134.4	Serbian	7666.2	Odia	543.2	Eastern Mari	22.9
Polish	146001.9	Latvian	7411.5	Tibetan	531.6	Luxembourgish	18.4
Vietnamese	139298.4	Malayalam	5815.1	Amharic	513.0	Uzbek	15.3
Dutch	135078.1	Mongolian	5777.3	Kyrgyz	489.5	Chechen	13.9
Arabic	110728.5	Gujarati	5593.9	Esperanto	475.1	Malagasy	11.2
Portuguese	105065.0	Nepali	4950.5	Lao	472.3	Low German	10.7
Greek	95750.9	Armenian	4884.7	Assamese	412.2	Mingrelian	6.1
Persian	93225.0	Macedonian	4745.4	Bashkir	363.9	Bishnupriya	5.4
Thai	91968.7	Marathi	4478.3	Welsh	333.1	Newari	4.0
Czech	76987.1	Telugu	3873.8	Pashto	261.7	Minangkabau	3.8
Turkish	72207.2	Urdu	3761.3	Galician	255.9	Egyptian Arabic	3.7
Swedish	50001.1	Kazakh	3325.4	Uyghur	219.8	Norwegian Nynorsk	3.7
Romanian	45590.6	Albanian	3224.9	Divehi	200.2	Turkmen	3.3
Ukrainian	44746.7	Khmer	3155.2	Kurdish	174.2	Piedmontese	3.1
Bulgarian	44118.5	Azerbaijani	3038.3	Yiddish	171.8	Malay	2.6
Finnish	41143.7	Burmese	3035.4	Sindhi	131.7	Goan Konkani	2.3
Korean	38158.4	Sinhala	2599.3	Western Panjabi	105.8	Latin	2.0
Hindi	32615.5	Norwegian	2583.2	Western Frisian	70.5	Lojban	1.5
Indonesian	23416.0	Kannada	2574.4	Sakha	68.8	Maltese	1.3
Slovak	21460.7	Belarusian	2339.5	Irish	63.2	Swahili	1.0

Table 5. The 37 programming languages considered in §4. Data is sourced from the Github split of RedPajama, and classified into a language based on the file extension.

Language	Size (in MB)	Language	Size (in MB)
Java	29493.0	Haskell	547.6
JavaScript	27910.4	TSQL	489.5
HTML	25864.1	Lua	393.4
XML	18804.0	Dockerfile	272.7
C++	15543.1	Makefile	265.7
Python	12970.1	TeX	256.9
Smalltalk	11580.5	XPixMap	248.7
Objective-C	10909.5	PowerShell	240.7
PHP	9837.4	CMake	118.5
Go	6287.2	Raku	106.9
Markdown	6137.3	Hack	79.1
C	6045.0	Julia	72.3
CSS	4084.9	Batchfile	60.9
Ruby	3381.4	Pod6	46.6
Scala	1376.8	FortranFreeForm	40.8
Smali	978.3	Fortran	31.2
reStructuredText	891.4	Motorola68KAssembly	22.7
VisualBasic.NET	563.0	Perl	2.0
Shell	551.6		

Table 6. The 5 domains considered in §4. Data is sourced from RedPajama.

Domain	Size (in MB)
Web	305139.9
Code	196506.0
Books	104975.0
Academic	89044.9
Wikipedia	20505.8