Fishing for Magikarp: Automatically Detecting Under-trained Tokens in Large Language Models

Anonymous ACL submission

Abstract

The disconnect between tokenizer creation and model training in language models has been known to allow for certain inputs, such as the infamous `_SolidGoldMagikarp` token, to induce unwanted behaviour. Although such 'glitch tokens' that are present in the tokenizer vocabulary, but are nearly or fully absent in training, have been observed across a variety of different models, a consistent way of identifying them has been missing. We present a comprehensive analysis of Large Language Model tokenizers, specifically targeting this issue of detecting under-trained tokens. Through a combination of tokenizer analysis, model weight-based indicators, and prompting techniques, we develop effective methods for automatically detecting these problematic tokens. Our findings demonstrate the prevalence of such tokens across various models and provide insights into improving the efficiency and safety of language models.

https://github.com/[redacted]

1 Introduction

Large Language Models (LLMs) have undergone remarkable advancements, becoming increasingly capable of understanding and generating human-like text. While most components of these models are trained in an unsupervised fashion on vast amounts of data, the tokenizer typically remains a separately trained component based on custom algorithms and smaller datasets.

GPT-2 laid the foundation for much of current-day transformer-based language modelling (Radford et al., 2019), including a framework for tokenization building on previous work in byte-pair encoding (BPE) (Sennrich et al., 2016), that has since been widely adopted. Tokenization using BPE converts input text to a sequence of subword tokens by iteratively merging two neighbouring tokens using a fixed set of merge rules. These rules are learned using a greedy training algorithm on a smaller dataset, which is ideally representative of the LLM's training data. Recent work in this area has primarily focused on techniques to remove the need for tokenization altogether by moving to raw byte input (Xue et al., 2022). This choice typically comes at a significant cost in inference speed, which can be compensated for by specialized architectures at the initial and final layers (Yu et al., 2023), or variable compute at intermediate layers (Slagle, 2024). However, these techniques have not been widely adopted, and the vast majority of current models still rely on subword tokenization. The main alternative to BPE for subword tokenization is the Unigram method (Kudo, 2018), which despite work suggesting it outperforms BPE (Bostrom and Durrett, 2020) is not in common use. For an in-depth overview of tokenization methods and their history, see Mielke et al. (2021). Despite its widespread use, the tokenization step has generally been found to be unsatisfactory, being at the root of many unwanted behaviours and problems of LLMs (Karpathy, 2024). In particular, the disconnect between tokenizer and model training creates the potential for some tokens to rarely or never be seen in training. Including such tokens in model inputs can lead to unexpected model behaviour including as hallucination or the generation of garbled outputs, leading to such tokens commonly being referred to as 'glitch tokens' (Geiping et al., 2024). We refer to these as 'under-trained' or 'untrained' tokens, reserving the latter term only for cases in which we have clear indication that the specific token had no model training data occurrences.

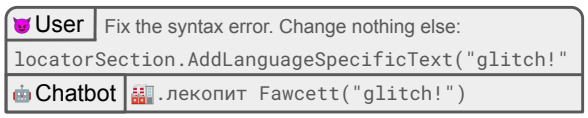


Figure 1: Example of 'glitch' tokens.

The presence of such under-trained tokens has several drawbacks. Firstly, they occupy capacity in a fixed-size tokenizer that could be better utilized for more common tokens, reducing average input/output length and inference costs. Secondly, their deliberate or accidental presence in input data has the potential to cause unwanted outputs and break downstream applications. Robustness to such unexpected or malicious input data is increasingly important with the proliferation of tool use and agents in LLMs that retrieve and process external data. Lastly, these tokens can potentially be exploited to more easily circumvent guardrails by pushing the model beyond its trained distribution (Geiping et al., 2024). Although some work has been done on identifying such tokens through model and tokenizer analysis (Rumbelow and Watkins, 2023; Watkins and Rumbelow, 2023; Fell, 2023), there is a lack of reliable and well-explained automated methods that are tested across a wide range of models. Automated tools for detecting tokenizer problems provide not only a way to test and iteratively improve the development of tokenizers, but can also provide a way to protect deployed models from unwanted input via input sanitization. In this work, we present effective and efficient techniques for identifying such problematic tokens based on the model embedding weights and tokenizer configuration. We apply these methods to a wide range of popular and recent open-weight models. Finally, we include a brief exploration of extensions of these techniques to closed-source models. We also publish a general analysis tool compatible with Hugging Face models, along with detailed results for each analyzed model.

2 Methods

Our method consists of three steps; i) first, we perform a tokenizer analysis by inspecting its vocabulary and observing its encoding/decoding behaviour, ii) second, we calculate a number of indicators that identify candidate under-trained tokens, and iii) third, we verify whether identified candidate tokens are indeed out of distribution by prompting the the target model.

2.1 Tokenizer analysis

We start by defining a number of useful categories for tokens. PARTIAL UTF-8 SEQUENCES are token representing sequences of bytes that can not be converted to Unicode characters, due to containing only part of a UTF encoding for a character. This is typical for 'fallback byte' tokens in the `0x80-0xFF` range, but depending on whether BPE was applied directly to bytes, can also include tokens with other partial Unicode characters.

UNREACHABLE TOKENS are those which are never the result of tokenizing text. We test this by checking if decoding the token to a string, and re-tokenizing this string, results in the token id. Such tokens are typically the result of tokenizer configuration errors or conflicts between trained and manually added vocabulary. As this test does not work when tokens can not be decoded to a string, we exclude partial UTF-8 sequences from this category.

SPECIAL TOKENS are manually defined tokens that typically bypass the normal (pre-)tokenization pipeline, and often carry specific meanings as control tokens, such as `<s>`. We identify special tokens using the patterns `<...>` and `[...]` and list them separately from unreachable tokens, even if they might be considered as such due to input sanitization in tokenizer preprocessing.

We detect and exclude partial UTF-8 sequences and unreachable tokens from our under-trained token detection pipeline, as they are not suitable for automatically building verification prompts. Our published model reports include tables with such tokens, and we briefly discuss some interesting model-specific results in section 3.2.

2.2 Under-trained token indicators

This section outlines our model architecture-dependent indicators, which we use to identify potentially under-trained tokens. An key distinction is made based on whether or not a model uses 'tied' embeddings (Inan et al., 2017), i.e. uses the same matrix for its input embeddings E_{in} and the 'output embeddings' matrix E_{out} in the final 'language modelling head' layer.

Regardless of the use of tied embeddings, all weights of the output embeddings influence the token predictions at every training step. All untrained tokens will see similar updates in training, 'moving away' from the mean output vector of the model (Biś et al., 2021)[1]. Alternatively, we can expect the model to learn to include a constant direction in its outputs, regardless of context (e.g. via the residual stream), to consistently output highly negative logits for tokens that are never a correct

[1]We assume the common setup with no bias term.

prediction. Thus, we can expect to find that under-trained token embeddings share a similar direction in output embedding space, and can identify them by using the distance to the embeddings of reference untrained tokens.

When embeddings are not tied, input embeddings for tokens which do not appear in the input for a training step are only affected by a potential weight decay term. If such a term is applied to the input embedding matrix, those embeddings corresponding to under-trained tokens will decay to zero over training. Alternatively, they will stay at a (typically low) initial value. The norm of the input embeddings thus provides an additional indicator for under-trained tokens with potentially higher sensitivity, and which does not require a set of known untrained tokens. Specifically, we expect it will not predict control tokens such as <s> that are only seen in inputs.

We use the norm of the input embeddings, and the cosine distance between output embeddings as our default under-trained token indicators for models with untied and tied embeddings, respectively. In addition we calculate and visualize a number of different indicators and correlation between them, including the Euclidean distance between output embeddings. More formal definitions and experiments with more complex output embedding indicators are outlined in Appendix A.

2.3 Verification of candidate tokens

Our proposed indicators naturally provide a ranking of candidate under-trained tokens, but do not give a definitive threshold, and their relative simplicity is likely to result in a somewhat noisy relation between indicator and model behaviour. To confirm that candidate tokens indeed induce unwanted model outputs, we verify all tokens which rank among the most likely 2% according to the chosen indicator, excluding partial UTF-8 sequences and unreachable tokens. This verification process involves constructing specific repetitive prompts that induces a high output probability for normal tokens, and checking if a candidate token has a very low ($< 1\%$) output probability. See Appendix B for model prompts.

2.4 Effectiveness of token indicators

This section validates our proposed indicators by relating them to both model behaviour, and training data statistics Although such training data statistics are rarely available, the we are able to do such a three-way comparison on the open OLMo v1.7 model (Groeneveld et al., 2024). Figure 2 shows a strong correlation between all proposed indicators and training data, not only predicting under-trained tokens, but extending to the entire range of token frequencies. Applying our verification step to all tokens, shows that despite their relative simplicity, our indicators are highly predictive of the maximal probability of token prediction (Figure 3). More precisely, 191 out of 49,575 tokens pass our verification step, compared to 175/993 when testing only the top 2% candidate tokens, showing the 2% threshold is a reasonable trade-off between computational cost and detecting the majority of highly under-trained tokens. Finally, Figure 4 shows examples of the visualizations we perform on all model indicators. These show a clear secondary peak near zero across models, as well as high correlation between alternative indicators, further validating their effectiveness.

3 Results

In this section, we present a summary of our key findings regarding under-trained token detection. Table 1 presents verification statistics and example verified tokens for a wide range of models. The number of verified under-trained tokens varies significantly across different model families and tokenizer vocabulary size, as well as depending on the number of unused special tokens a model's tokenizer allows as plain-text input. The percentage of verified tokens typically ranges between 5–50% of tested candidate tokens, corresponding to 0.1–1% of the total vocabulary.

Given the model-specific nature and the extensive volume of results, we discuss some common findings as well as showcase some representative examples for particular models. Detailed reports covering an increasing number of tested models and token types are available in our repository.

3.1 Common observations

Although many of our findings are dependent on model-specific details such as tokenizer training and configuration, model architecture, and training data, there are a number of commonalities that appear across many different models.

3.1.1 Single-byte tokens

Tokens representing a single byte are a common source of untrained tokens. The most common occurrence are the bytes `0xF5`–`0xFF` which are not

Model	#Tokens	Tied Emb.	#Confirmed	Examples
GPT-2 Medium (0.4B)	50,257	Yes	49/999	`InstoreAndOnline` `reportprint` `_externalToEVA`
GPT-2 XL (1.5B)	50,257	Yes	67/999	`InstoreAndOnline` `_RandomRedditor` `embedreportprint`
GPT-J 6B	50,400	No*	200/999	`_attRot` `_externalToEVA` `_SolidGoldMagikarp`
Phi-2 (2.7B)	50,295	No*	103/999	`DragonMagazine` `_TheNitrome` `_SolidGoldMagikarp`
Pythia 6.7B	50,277	No	14/993	`FFIRMED` `_taxp` `_affidav`
GPT-NeoX 20B	50,277	No	10/993	`FFIRMED` `_taxp` `_affidav`
OLMo v1.7 7B	50,280	No	178/993	`_ǧ\\[` `medscimonit` `FFIRMED` `_[****`
Llama2 7B	32,000	No	20/639	`_Mediabestanden` `_Portály` `oreferrer`
Llama2 70B	32,000	No	32/639	`_Mediabestanden` `_Portály` `ederbörd`
Mistral 7B v0.3	32,000	No	53/637	`\uefc0` `});\r` `ɕ` `>?[<` `_febbra` `_uitgen`
Mixtral 8x7B	32,000	No	44/637	`\uefc0` `_/**\r` `ɕ` `];\r`
Rakuten 7B	48,000	No	66/957	`\uefc0` `_/**\r` `ɕ` `_febbra` `稲田大学`
Qwen1.5 32B	151,646	No	2450/2966	`_ForCanBeConvertedToF` `(stypy` `$PostalCodesNL`
Qwen1.5 72B Chat	151,646	No	2047/2968	`_ForCanBeConverted` `useRalative` `_typingsJapgolly`
StableLM2 12B	100,288	No	138/1997	`_ForCanBeConverted` `\tTokenNameIdentifier` `_StreamLazy`
Llama3 8B	128,256	No	556/2540	`_ForCanBeConverted` `ŬыцNŬыцN` `_CLIIIK` `krvldkf` `글상위`
Llama3 70B	128,256	No	462/2540	`$PostalCodesNL` `итися` `ılmaktadır` `ーション` `;\r\r\r\n`
Command R (35B)	255,029	Yes	306/5012	`AddLanguageSpecificText` `_ARStdSong` `目前尚未由人工引`
Command R+ (104B)	255,029	Yes	75/5012	`AddLanguageSpecificText` `tocguid` `ephritidae`
Gemma 2B	256,000	Yes	3161/5117	`हिंदीखरीदारी` `ˆ(@)$_` `_coachTry` `_AcceptedLoading` `ICTOGRAM`
Gemma 7B	256,000	Yes	800/5117	`हिंदीखरीदारी` `EnglishChoose` `_queſto` `_stockfotografie` `⊠`
Starcoder2 15B	49,152	No	128/968	`ittrLoremipumdolorsitametconsecteturadipiscingelitIntegervelvel`
Yi 9B	64,000	No	245/1278	`\\+::\\+` `mcited` `mabaochang` `nConsequently`
Jamba v0.1 (52B)	65,536	No	6/1280	`derrelsc` `]{}]{}` `ronicsystems`

Table 1: **Detection of under-trained tokens.** #Confirmed are the confirmed/tested numbers for the tokens tested in verification that are predicted with a maximal probability of $< 1\%$ across verification prompts. Examples were manually chosen for readability, similarity across models or for being particularly striking. Note that the leading '_' in tokens such as `_SolidGoldMagikarp` indicates a leading space.

*These models include a bias in their final layer, which does not affect our results as we use their input embeddings.

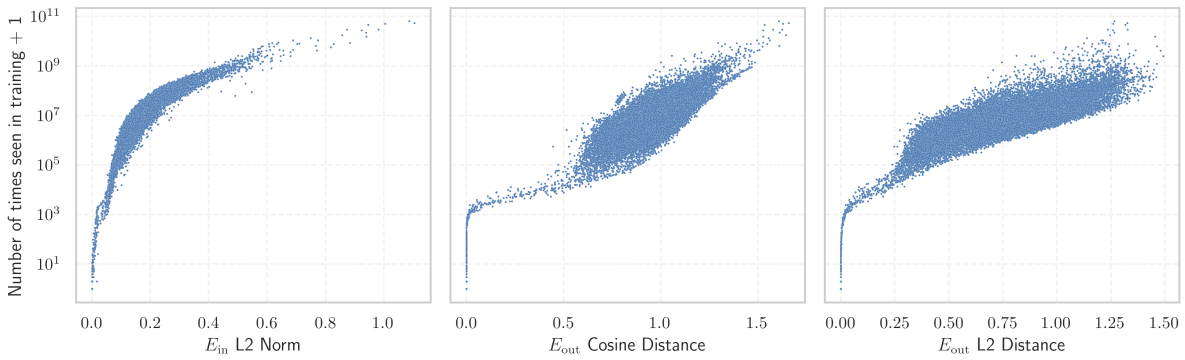


Figure 2: Under-trained token indicators are predictive of training data. The embedding based under-trained token indicators for the OLMo v1.7 7B model and the number of times each token appears in the first epoch of the training data are shown. All indicators correlate highly with the number of times a token is seen in training, not only at the expected lower values, but extending across ten orders of magnitude.

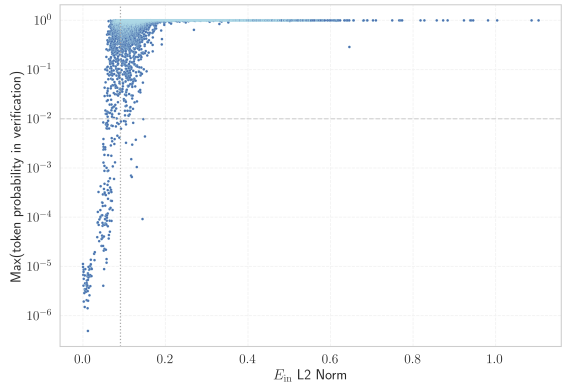


Figure 3: Under-trained token indicator are predictive of verification probability. The rate of successful verification ($p < 0.01$) correlates very highly with our proposed indicator, with no false positives at low values of the indicators and a low rate of false negatives. The dotted line indicates the default 2% threshold used for verification.

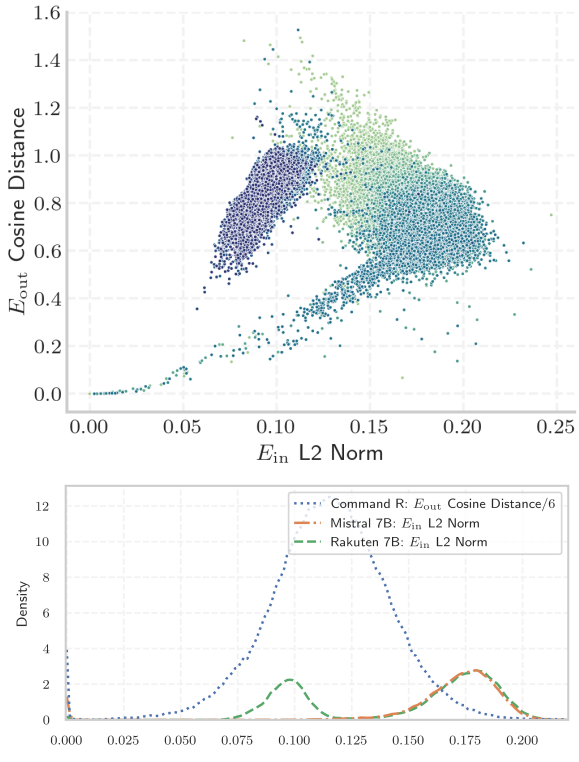


Figure 4: Comparison of indicators. The scatter plots are coloured by token id, from light green to dark blue. Top: Rakuten 7B showing a separate cluster for added tokens, and high correlation near zero. Bottom: In density plots, a clear peak appears near zero for most models, giving rise to a bimodal distribution.

used in UTF-8 encoded text[2], and are a convenient source for quickly locating reference untrained tokens for indicators which require them. In addition, many tokenizers including from the Gemma, Llama2 and Mistral families include every byte as a token, with many of them in the normal ASCII range `0x00`–`0x7F` being redundant and unreachable due to the existence of a token for the corresponding character. These issues are not universal, and we also find models which include precisely the 243 bytes used in UTF-8 as tokens. Untrained single byte tokens are typically classified as 'partial UTF-8 sequences' or 'unreachable', and our indicators are effective in revealing which ones are never or rarely seen in training.

3.1.2 Intermediate BPE fragments

All tested models use BPE based tokenization, which retains the original tokens after a merge, often causing intermediate 'junk' tokens (Bostrom and Durrett, 2020). When detecting these as under-trained, will denote the more complete token in parentheses, e.g. `_TheNitrome` (`_TheNitromeFan`) in the GPT-2 tokenizer. In some occasions the longest token is *also* under-trained, alongside a variety of fragments. The same mechanism appears to explain many under-trained partial UTF-8 sequences in byte-level BPE tokenizers, with multiple bytes being merged over several steps, leaving potentially many intermediate tokens with partial Unicode characters.

3.1.3 Special tokens

Many models include untrained special tokens, such as `<pad>`, `<unk>`, or `<|unused_123|>`. In the following discussion we generally omit mentioning them, unless their status as an (un)trained token is particularly surprising, as their inclusion in the tokenizer and training data is typically deliberate, for purposes such as the ability to fine-tune models without changing tokenizers. One common observation is that on many occasions tokens such as `<mask>`, which we expect to be completely untrained, nevertheless appear to have been seen in training. A likely source for this is code repositories or guides about language models using these tokens in normal text, along with tokenizers allowing such special control tokens in input text.

Special tokens can be unreachable due to input sanitization as well as configuration errors. In particular, both the Gemma and Yi models include

[2]See Appendix C for a primer on UTF-8 encoding.

special tokens relating to HTML tags, which were initially detected as unreachable, with the tags being split up in pre-tokenization[3].

3.2 Model-specific observations

In this section we outline some model-specific observations, grouped by the tokenizer used. These examples are mainly intended to illustrate the variety of different under-trained tokens and configuration issues that can be found using our methods, and are not exhaustive.

GPT-2 (Radford et al., 2019) introduced the framework for much of current-day LLMs, and the tokenizer has been re-used extensively. We confirm previous findings with a significant number of tokens related to (fragments of) usernames (e.g. `_TheNitrome`, `_RandomRedditor`). We also find a number of under-trained non-English tokens. We also detect that all ASCII control characters except for the newline character, but including the tab and carriage return characters, appear untrained. This suggests a potential mismatch in data normalization between training and inference.

GPT-J 6B (Wang and Komatsuzaki, 2021) and **Phi-2** (Microsoft, 2023) are independent models which both also use the GPT-2 tokenizer, and have significantly more under-trained tokens, likely due to their training data being further removed from the data used to train the tokenizer. These additional tokens include `_SolidGoldMagikarp`, which is not among verified candidates in GPT-2.

GPT-NeoX is an open-source library and associated family of models which uses a tokenizer with the same vocabulary size as GPT-2, but trained on the same 'The Pile' dataset also used for model training, and with added tokens for multiple spaces (Black et al., 2022). The GPT-NeoX 20B model has very few under-trained tokens, likely in part due to this alignment between tokenizer and model training, with the fragment `FFIRMED` showing up most consistently. The Pythia 6.7B model based on the same library (Biderman et al., 2023) also shows very similar results.

OLMo open language models (Groeneveld et al., 2024) also use the GPT-NeoX tokenizer, but have a much higher rate of under-trained tokens, including a wide range of punctuation-based tokens. We also detect over 200 unreachable tokens representing combinations of spaces and line breaks in the tokenizer, which appear to be caused by the aforementioned 'multiple spaces' tokens taking precedence. However, many of them appear to have been seen in training, based on both our indicators and training data statistics[4].

Furthermore, we noticed that embedding based indicators are not near zero for the GPT-NeoX and Pythia models, as well as v1 of the OLMo model. For the GPT-NeoX/Pythia models, this was explained by a specific implementation of weight decay, where only weights that are used in the forward pass are affected, but we find that having low but non-zero embeddings is still a good predictor for under-trained tokens. The OLMo v1 model instead applies no weight decay, and requires using output embedding based indicators instead. However, the OLMo v1.7 model does apply weight decay to embeddings, and its embedding norms are near zero for untrained tokens (cf. Figure 2), and we use only this more recent version in this work.

Llama2 models (Touvron et al., 2023) use an relatively compact BPE tokenizer, and have a low number of under-trained tokens, mostly relating to long non-English words, including `_Mediabestanden`, `_Расподела`, and `_Portály`. We also find under-trained intermediate fragments such as `_gepublic` (`_gepubliceerd`). Several of these tokens were also found in previous work on steering model outputs (Geiping et al., 2024).

Mistral models (Jiang et al., 2023, 2024) use a similar tokenizer, but its vocabulary includes a significant number of multi-character punctuation sequences ending in a carriage return (`\r`), which are the main source of under-trained tokens. The `\uefc0` token representing a single unassigned Unicode character in the 'private use area' is consistently among the most under-trained, along with ᦶ, a character from the Limbu script.

Rakuten 7B (Rakuten Group et al., 2024) is a derived model with an extended vocabulary for Japanese, and continued pre-training. Among the extended vocabulary we find a few under-trained fragments such as 稲田大学 (早稲田大学, 'Waseda University'). Their presence is proportional to the extended vocabulary, which forms a distinct cluster when visualising their indicators (see Figure 4).

Gemma is a family of models by Google Deepmind (Gemma Team et al., 2024) and uses a large 256,000 token vocabulary, which includes a sig-

[3]The Gemma team released a fix in response to our report, and the 01.AI team advise not to use the 'fast' version. Our reported results are based on the latest recommended versions.

[4]This was in part traced to a breaking change in `tokenizers` v0.14 (Luca Soldaini, personal communication).

nificant number of under-trained fragments in various scripts. Most notably we find many under-trained tokens which contain 'ſ' (an archaic form of 's' in German), including _müſſen, as well as a number of translations of 'stock photos' such as _stockbilder and _stockfotos.

Command R/R+ are models by Cohere (2024) which also have a large multi-lingual vocabulary with over 255,000 tokens. The most notable discovery in these models was that over 1,400 manually added tokens of emojis are categorized as unreachable, and are all clearly untrained according to their indicators. Additionally, among partial UTF-8 sequences are several tokens related to the English flag followed by invisible Unicode 'tag' characters, which we tracked to a conversion step from image-based flags to emojis in an open-source pipeline for parsing Wikipedia pages, potentially affecting other models as well.[5]

The **tiktoken** library by OpenAI (OpenAI, 2024), includes the 'cl100k' tokenizer as used in GPT-3.5/GPT-4 as well as several other models. This tokenizer use a pre-tokenization pattern which allows not only a starting space, but many other single punctuation characters at the start of a token. This choice results in tokens such as `\tTokenNameIdentifier` and `$PostalCodesNL`, which are highly sensitive to pre-tokenization splitting, with leading spaces before the token resulting in different tokenization. In combination with their specific content, this is likely to have made them more severely under-trained across models.

StableLM2 is a model by Stability AI (Bellagente et al., 2024) which uses a slightly modified version of this tokenizer. Due to the addition of digit splitting, the original multi-digit tokens were expected to show up as both unreachable and untrained, but were initially only detected as untrained due to a tokenizer configuration error[6].

Qwen is a model family by Alibaba (Bai et al., 2023) which significantly extends the 'cl100k' tokenizer to over 150,000 tokens. The added tokens and large inherited tokenizer results in many under-trained tokens, and among added tokens we find archaic Chinese characters (such as 骁) and Korean characters which are typographically valid but never seen in normal text (such as 랐).

Llama3 is a recent model family by Meta AI (2024) which also extends this tokenizer with 28,000 additional tokens. Aside from sharing many under-trained tokens with other models using this tokenizer, the newly added tokens include additional under-trained tokens such as ŬыцNŬыцN and `krvldkf`.

StarCoder2 is a family of models resulting from the BigCode project, an open-scientific collaboration focused on code (Lozhkov et al., 2024). The open nature of the project represents a great opportunity for further investigation, allowing us to determine the source of under-trained tokens in the published tokenizer training data. We find a single document which illustrates maximal variable lengths in Java by repeating 'LoremipumdolorsitametdconsecteturadipiscingelitIntegervelvelittr' as the source of several long under-trained tokens, a single document with base-64 encoded strings as the origin of tokens such as `BjKPZFq`, and a single source code file with a list of solutions of a German Wordle game with words categorized by dialect as the source of several tokens such as `Ostschwizertütsch` relating to Swiss German dialects. Furthermore, the tokenizer is unique in missing the `0xF1` byte as a token in addition to not including unused UTF-8 bytes, and input text containing this byte results in `<|endoftext|>` being used as a fallback 'unknown' token.

Yi 9B is a base model by 01.AI whose training data is focused on English and Chinese (01.AI et al., 2024). Most notable among results are a number of strange tokens starting with 'n', including `nConsequently` and `nInterestingly` which may have been caused by incorrectly processing newline characters in tokenizer training data. In addition, three tokens with Chinese phrases including 毛泽东 are unusual unreachable tokens.

Jamba v0.1 is a model from AI21 based on a hybrid Transformer-Mamba mixture-of-experts architecture with 52B total parameters (Lieber et al., 2024). This model has very few tokens that pass our strict threshold for verification, and probabilities for token output are often unusually close to one. Tokenizer analysis does reveal 1,542 untrained special tokens, with `<|startoftext|>` as the only special token which has seen training. The latter is also an extreme outlier in our verification, with our indicators showing it to be clearly seen in training, while the maximal probability of outputting the token is $\approx 10^{-8}$. The unusually sharp probability distributions may be an effect of the novel architecture of this model.

[5] Our fix [link redacted] for this has been released.

[6] This bug was fixed by disabling the 'slow' tokenizer.

4 Application to closed-source models

As our techniques involve using the model weights, they are not directly applicable to closed-source models. However, the experience gained in inspecting a large variety of open models has provided insight which may transfer to closed models. For these tests, we use a custom prompt designed to exactly repeat strings and see if models appear incapable of doing so. Using these techniques we can readily identify under-trained tokens in closed OpenAI and Mistral models by using open models which share a tokenizer, as well as in the Claude 2.1 model by only using the tokenizer to identify candidate tokens. See Appendix D for details.

5 Discussion

The presence of under-trained tokens has several negative consequences for language models, including inefficient inference and the potential to bypass guardrails. Our investigation has shown a wide variety of untrained and under-trained tokens present in model tokenizers. Even with our relatively strict threshold for verification, we detect the presence of such tokens across all tested models, with typically around 0.1–1% of the vocabulary consisting of severely under-trained tokens, although their prevalence varies significantly. The most important factors in a model having many under-trained tokens, aside from simply having a large vocabulary, appears to be whether the tokenizer was trained on similar data as the model. Models which re-use a large external tokenizer, and then train from scratch, are among those with the highest number of under-trained tokens.

Analyzing the tokenizer directly can detect several of these without the need for any training, including unreachable tokens which do not encode back to their representation, and unused byte fallback tokens. This can be particularly useful in quickly catching tokenizer configuration errors, which appear to be particularly common when custom vocabulary is manually added. Additionally using the model embedding weights directly is a reliable way to detect tokens which are under-trained, although the care should be taken to take into account the model architecture.

Based on our findings, we can summarize a number of recommendations within the scope of current tooling. Firstly, ensure input data pre-processing is identical across tokenizer training data, model training data, and model inference. In particular, consider carefully how to handle carriage returns, tab characters, and special tokens present as plain text in training data and user input. Secondly, carefully consider tokenization training data, ensuring it is representative of model training data. Finally, after training a tokenizer, check for unreachable tokens by encoding and decoding the vocabulary to ensure manually added tokens are handled correctly. Additionally, when training a base model, check for under-trained tokens after smaller test runs, or test on a different corpus, to reveal pre-processing bugs that cause unrepresentative inputs in the main training data.

In addition to providing a set of useful tools for improving models and tokenizers, our work suggests several directions for future research. Firstly, the results from StarCoder2 (section 3.2) highlight a potential weakness in BPE training in that occurrences in a single document or repository are able to define a token by themselves. Strategies for preventing this, such as limiting the count for pairs to be merged by document, should be explored to prevent this. Secondly, we note that byte-based BPE produces more intermediate fragments which additionally have the ability to cause outputs to be undecodable. The trade-off between more efficient encoding and these downsides is particularly under-explored. Although allowing such tokens may lead to lower average token counts, it also leads to more untrained 'fragments' and tokens which are less semantically meaningful. Techniques such as BPE-dropout (Provilkov et al., 2020) have been proposed to compensate for under-trained intermediate fragments, but direct comparisons on state-of-the-art models are lacking. Finally, we noticed differences between models in terms of how they apply weight decay to tokens not present in input, including not applying weight decay to embeddings, applying it only to tokens seen in a batch, or applying it to all weights. This choice may affect how well models remember the meaning of rare tokens and likely mitigate the severity and impact of under-trained tokens. Although this choice has been known to be important in older models (Sedhain et al., 2015), we are not aware of systematic ablations in recent LLMs.

In conclusion, our findings highlight a range of tokenizer issues, and the severity of these varies across different models. By analyzing tokenizers and model embeddings, we can identify under-trained tokens and improve the efficiency and security of LLMs.

6 Limitations

Although our pipeline for finding under-trained tokens is effective at finding such tokens in a wide range of models, it still has a number of significant limitations.

Most notably, the output embedding based indicators require manually specifying a set of reference under-trained tokens, preventing the method from being fully automated for the minority of models with tied embeddings. Secondly, the output embedding based indicators are heuristic, and based on a hypothesis for the internal representation and training dynamics. Further research into model interpretability could refine our understanding of such representations, and lead to more effective indicators. The input embeddings based indicator, while not requiring such manual input, is only applicable to models without tied embeddings, and depends on particular choices for weight decay and/or initialization. Although this constitutes the majority of models, there are a significant number of exceptions, and the exact weight decay used is often not well documented.

Aside from limitations in the ability to automatically calculate under-trained token indicators, the relation between our proposed indicators and model behaviour is noisy. Both the indicators themselves, as well as the verification results, can be more indicative of problematic model behaviour on different occasions. Specifically, there are certain cases where the indicators we use offer a more reliable guide of a token's tendency to induce unwanted output in typical prompting compared to our verification prompting techniques. These cases include input/output asymmetry, where tokens are solely present as inputs (e.g., <BOS>), or situations where the model exhibits a strong bias towards English, consistently producing translated outputs. Another common occurrence is output of the equivalent token without a leading space, although the variation in our verification prompts compensates for this. On the other hand, there are cases where tokens are rejected by the verification process, but can still induce incorrect behaviour, mainly due to our strict threshold and repetitive verification prompts, which are aimed at detecting the most reliable under-trained tokens. However, despite these limitations, verification using prompting is highly effective in identifying a threshold below which candidate tokens induce unwanted behaviour, and selecting the most effective candidate tokens.

Finally, the scope of our work is limited by focusing exclusively on models which use byte-pair encoding based tokenization. Results for Unigram based models are likely to be significantly different, with both the lack of intermediate fragments, and randomized tokenization preventing the intermediate fragments which are a source of under-trained tokens.

References

01.AI, Alex Young, Bei Chen, Chao Li, Chengen Huang, Ge Zhang, Guanwei Zhang, Heng Li, Jiangcheng Zhu, Jianqun Chen, Jing Chang, Kaidong Yu, Peng Liu, Qiang Liu, Shawn Yue, Senbin Yang, Shiming Yang, Tao Yu, Wen Xie, Wenhao Huang, Xiaohui Hu, Xiaoyi Ren, Xinyao Niu, Pengcheng Nie, Yuchi Xu, Yudong Liu, Yue Wang, Yuxuan Cai, Zhenyu Gu, Zhiyuan Liu, and Zonghong Dai. 2024. Yi: Open foundation models by 01.AI. *Preprint*, arXiv:2403.04652.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen technical report. *Preprint*, arXiv:2309.16609.

Marco Bellagente, Jonathan Tow, Dakota Mahan, Duy Phung, Maksym Zhuravinskyi, Reshinth Adithyan, James Baicoianu, Ben Brooks, Nathan Cooper, Ashish Datta, Meng Lee, Emad Mostaque, Michael Pieler, Nikhil Pinnaparju, Paulo Rocha, Harry Saini, Hannah Teufel, Niccolo Zanichelli, and Carlos Riquelme. 2024. Stable LM 2 1.6B technical report. *Preprint*, arXiv:2402.17834.

Stella Biderman, Hailey Schoelkopf, Quentin Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, Aviya Skowron, Lintang Sutawika, and Oskar van der Wal. 2023. Pythia: A suite for analyzing large language models across training and scaling. *Preprint*, arXiv:2304.01373.

Daniel Biś, Maksim Podkorytov, and Xiuwen Liu. 2021. Too much in common: Shifting of embeddings in transformer language models and its implications. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for*

Computational Linguistics: Human Language Technologies, pages 5117–5130.

Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. 2022. GPT-NeoX-20B: An open-source autoregressive language model. *Preprint*, arXiv:2204.06745.

Kaj Bostrom and Greg Durrett. 2020. Byte pair encoding is suboptimal for language model pretraining. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4617–4624, Online. Association for Computational Linguistics.

Cohere. 2024. Cohere Command R documentation.

Martin Fell. 2023. A search for more ChatGPT / GPT-3.5 / GPT-4 "unspeakable" glitch tokens. Blog post.

Jonas Geiping, Alex Stein, Manli Shu, Khalid Saifullah, Yuxin Wen, and Tom Goldstein. 2024. Coercing llms to do and reveal (almost) anything. *Preprint*, arXiv:2402.14020.

Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, Léonard Hussenot, Aakanksha Chowdhery, Adam Roberts, Aditya Barua, Alex Botev, Alex Castro-Ros, Ambrose Slone, Amélie Héliou, Andrea Tacchetti, Anna Bulanova, Antonia Paterson, Beth Tsai, Bobak Shahriari, Charline Le Lan, Christopher A. Choquette-Choo, Clément Crepy, Daniel Cer, Daphne Ippolito, David Reid, Elena Buchatskaya, Eric Ni, Eric Noland, Geng Yan, George Tucker, George-Christian Muraru, Grigory Rozhdestvenskiy, Henryk Michalewski, Ian Tenney, Ivan Grishchenko, Jacob Austin, James Keeling, Jane Labanowski, Jean-Baptiste Lespiau, Jeff Stanway, Jenny Brennan, Jeremy Chen, Johan Ferret, Justin Chiu, Justin Mao-Jones, Katherine Lee, Kathy Yu, Katie Millican, Lars Lowe Sjoesund, Lisa Lee, Lucas Dixon, Machel Reid, Maciej Mikua, Mateo Wirth, Michael Sharman, Nikolai Chinaev, Nithum Thain, Olivier Bachem, Oscar Chang, Oscar Wahltinez, Paige Bailey, Paul Michel, Petko Yotov, Pier Giuseppe Sessa, Rahma Chaabouni, Ramona Comanescu, Reena Jana, Rohan Anil, Ross McIlroy, Ruibo Liu, Ryan Mullins, Samuel L Smith, Sebastian Borgeaud, Sertan Girgin, Sholto Douglas, Shree Pandya, Siamak Shakeri, Soham De, Ted Klimenko, Tom Hennigan, Vlad Feinberg, Wojciech Stokowiec, Yu hui Chen, Zafarali Ahmed, Zhitao Gong, Tris Warkentin, Ludovic Peran, Minh Giang, Clément Farabet, Oriol Vinyals, Jeff Dean, Koray Kavukcuoglu, Demis Hassabis, Zoubin Ghahramani, Douglas Eck, Joelle Barral, Fernando Pereira, Eli Collins, Armand Joulin, Noah Fiedel, Evan Senter, Alek Andreev, and Kathleen Kenealy. 2024. Gemma: Open models based on Gemini research and technology. *Preprint*, arXiv:2403.08295.

Dirk Groeneveld, Iz Beltagy, Pete Walsh, Akshita Bhagia, Rodney Kinney, Oyvind Tafjord, Ananya Harsh Jha, Hamish Ivison, Ian Magnusson, Yizhong Wang, Shane Arora, David Atkinson, Russell Authur, Khyathi Raghavi Chandu, Arman Cohan, Jennifer Dumas, Yanai Elazar, Yuling Gu, Jack Hessel, Tushar Khot, William Merrill, Jacob Morrison, Niklas Muennighoff, Aakanksha Naik, Crystal Nam, Matthew E. Peters, Valentina Pyatkin, Abhilasha Ravichander, Dustin Schwenk, Saurabh Shah, Will Smith, Emma Strubell, Nishant Subramani, Mitchell Wortsman, Pradeep Dasigi, Nathan Lambert, Kyle Richardson, Luke Zettlemoyer, Jesse Dodge, Kyle Lo, Luca Soldaini, Noah A. Smith, and Hannaneh Hajishirzi. 2024. Olmo: Accelerating the science of language models. *Preprint*, arXiv:2402.00838.

Hakan Inan, Khashayar Khosravi, and Richard Socher. 2017. Tying word vectors and word classifiers: A loss framework for language modeling. *Preprint*, arXiv:1611.01462.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b. *Preprint*, arXiv:2310.06825.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2024. Mixtral of experts. *Preprint*, arXiv:2401.04088.

Andrej Karpathy. 2024. Let's build the GPT Tokenizer. YouTube Video.

Taku Kudo. 2018. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 66–75, Melbourne, Australia. Association for Computational Linguistics.

Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, Omri Abend, Raz Alon, Tomer Asida, Amir Bergman, Roman Glozman, Michael Gokhman, Avashalom Manevich, Nir Ratner, Noam Rozen, Erez Shwartz, Mor Zusman, and Yoav Shoham. 2024. Jamba: A hybrid transformer-mamba language model. *Preprint*, arXiv:2403.19887.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas KrauSS, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. Starcoder 2 and the stack v2: The next generation. *Preprint*, arXiv:2402.19173.

Meta AI. 2024. Introducing meta llama 3: The most capable openly available llm to date.

Microsoft. 2023. Phi-2: The surprising power of small language models.

Sabrina J. Mielke, Zaid Alyafeai, Elizabeth Salesky, Colin Raffel, Manan Dey, Matthias Gallé, Arun Raja, Chenglei Si, Wilson Y. Lee, Benoît Sagot, and Samson Tan. 2021. Between words and characters: A brief history of open-vocabulary modeling and tokenization in nlp. *Preprint*, arXiv:2112.10508.

OpenAI. 2024. tiktoken: a fast BPE tokeniser for use with OpenAI's models.

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

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Rakuten Group, Aaron Levine, Connie Huang, Chenguang Wang, Eduardo Batista, Ewa Szymanska, Hongyi Ding, Hou Wei Chou, Jean-François Pessiot, Johanes Effendi, Justin Chiu, Kai Torben Ohlhus, Karan Chopra, Keiji Shinzato, Koji Murakami, Lee Xiong, Lei Chen, Maki Kubota, Maksim Tkachenko, Miroku Lee, Naoki Takahashi, Prathyusha Jwalapuram, Ryutaro Tatsushima, Saurabh Jain, Sunil Kumar Yadav, Ting Cai, Wei-Te Chen, Yandi Xia, Yuki Nakayama, and Yutaka Higashiyama. 2024. Rakutenai-7b: Extending large language models for Japanese. *Preprint*, arXiv:2403.15484.

Jessica Rumbelow and Matthew Watkins. 2023. SolidGoldMagikarp (plus, prompt generation). Blog Post.

Suvash Sedhain, Aditya Krishna Menon, Scott Sanner, and Lexing Xie. 2015. Autorec: Autoencoders meet collaborative filtering. In *Proceedings of the 24th International Conference on World Wide Web*, pages 111–112. ACM.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. *Preprint*, arXiv:1508.07909.

Kevin Slagle. 2024. SpaceByte: Towards deleting tokenization from large language modeling. *Preprint*, arXiv:2404.14408.

The Unicode Consortium. 2023. The Unicode standard. version 15.0 core specification.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *Preprint*, arXiv:2307.09288.

Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model.

Matthew Watkins and Jessica Rumbelow. 2023. SolidGoldMagikarp III: Glitch token archaeology. Blog Post.

Linting Xue, Aditya Barua, Noah Constant, Rami Al-Rfou, Sharan Narang, Mihir Kale, Adam Roberts, and Colin Raffel. 2022. ByT5: Towards a token-free future with pre-trained byte-to-byte models. *Preprint*, arXiv:2105.13626.

Lili Yu, Dániel Simig, Colin Flaherty, Armen Aghajanyan, Luke Zettlemoyer, and Mike Lewis. 2023. MEGABYTE: Predicting million-byte sequences with multiscale transformers. *Preprint*, arXiv:2305.07185.

A Under-trained token indicators

This section gives more precise definitions of our propsosed under-trained token indicators we used, as well as experiments comparing some potential alternatives.

A.1 Definitions

To calculate the indicators based on the output embeddings E_{out}, we start by defining a set of known untrained or highly under-trained embedding indices t_{ref}, such as the token ids for tokens such as `<unused_token123>`, or the space of embeddings above the tokenizer vocabulary size.

Next, we calculate the mean unused token embedding vector

$$u_{\text{ref}} = \frac{1}{|t_{\text{ref}}|} \sum_{i \in t_{\text{ref}}} E_{\text{out},i}$$

Finally, we take the cosine distances $C(E_{\text{out}}, u_{\text{ref}})$ between this mean unused embedding vector and rows in E_{out}, where $C(A, x)$ is the vector of cosine distances between x and rows of A

$$C(A, x)_i = 1 - \frac{A_i \cdot x}{\|A_i\| \|x\|}$$

In addition to the cosine distance between output embeddings, we also calculate and visualize the the Euclidean distance between output embeddings to the untrained reference $L_2(E_{\text{out}} - u_{\text{ref}})$ where

$$L_2(A)_i = \|A_i\|$$

For models with tied embeddings, we use the cosine distance based indicator $C(E_{\text{out}}, u_{\text{ref}})$ to select candidate tokens. For models with tied embeddings, we use the norm of E_{in}, denoted $L_2(E_{\text{in}})$. In addition we calculate and visualize all output embedding based indicators.

A.2 Comparison of alternative indicators

For some models, in particular those in the Gemma series (Gemma Team et al., 2024), we noticed a very high similarity between the rows of their (tied) embedding matrix. Such similarity between embeddings has been noted before, and has been attributed to all embeddings being pushed in a common direction during training (Biś et al., 2021). Although a constant component in all output embeddings has no effect on model predictions, as softmax is invariant to a constant shift of all logits, such similarity may affect the effectiveness of our under-trained token indicators.

To compensate for this, we tested two variations for reducing or removing this constant component. Centering the embeddings by subtracting their mean, and removing their first principal component:

$$\hat{E}_{\text{out},i} = E_{\text{out},i} - \frac{1}{|E_{\text{out}}|} \sum_j E_{\text{out},j}$$

$$U = \text{PCA}(E_{\text{out}})$$

$$\tilde{E}_{\text{out},i} = E_{\text{out},i} - (U_1^T E_{\text{out},i}) U_1$$

We can then take the cosine distance between rows in these adjusted output embedding matrices to obtain the additional indicators $C(\hat{E}_{\text{out}}, \hat{u}_{\text{ref}})$ and $C(\tilde{E}_{\text{out}}, \tilde{u}_{\text{ref}})$. Testing additional indicators on a small range of models (see Table 2) shows no consistent improvement in using these more complex methods.

B Verification details

We use three repetitive prompts to induce models to output the candidate token we are testing, shown in Table 3.

These prompts are all designed to be suitable for base models and not require specialized instruction tuning or prompt templating. For each prompt we generate three tokens and check the maximal probability of our target token being predicted, and then take the maximum of this again over all three prompts. Variation in quoting and spacing helps to ensure we do not detect false positives based on models producing similar tokens without spaces, or tokens which start with punctuation partially merging with quotes.

C A short primer on UTF-8 encoding

UTF-8 is the most prevalent encoding scheme used to represent text in computers and communication protocols worldwide. It efficiently encodes Unicode characters, which encompass a vast range of characters from various writing systems and symbols (The Unicode Consortium, 2023). Encoding to UTF-8 is often the first step in tokenization.

UTF-8 encoding can be summarized as follows:

- ASCII (code points below 128): Single byte, binary 0xxxxxxx representing up to 7 bits.
- 2-byte sequences: 110xxxxx, 10xxxxxx representing up to 11 bits.

	Number of verified under-trained tokens using metric				
Model	$C(E_{\text{out}}, u_{\text{ref}})$	$C(\hat{E}_{\text{out}}, \hat{u}_{\text{ref}})$	$C(\tilde{E}_{\text{out}}, \tilde{u}_{\text{ref}})$	$L_2(E_{\text{out}} - u_{\text{ref}})$	$L_2(E_{\text{in}})$
Command R	306	302	301	309	n/a
Command R+	75	79	74	83	n/a
Gemma 2B	3161	3310	2889	3052	n/a
Gemma 7B	800	802	797	800	n/a
GPT-2 Medium	49	49	49	49	n/a
Mistral 7B v0.3	63	54	63	60	53
Olmo 7B v1.7	177	178	177	179	175

Table 2: Effectiveness of different indicators. For each under-trained token indicator, we verified the top 2% of tokens, and show the number of these that pass our 1% verification threshold. No consistent pattern is seen to justify the more complex alternatives.

Verification prompt #1.
<token> is replaced with the token we are testing.

```
 This device repeats its input exactly, as shown in these 100 identical trials.
Input: «<token>»
Output: «<token>»
<repeated 18 additional times>
Input: «<token>»
Output:
```

Verification prompt #2
<token> is replaced with the token we are testing.

```
 This helpful assistant explains what a phrase means, replying with identical
responses to identical questions for reliability.
User: What does 'edslfkj' mean?
Assistant: 'edslfkj' has no specific meaning I am aware of.
User: what does '<token>' mean?
Assistant:
```

Verification prompt #3
<token> is replaced with the token we are testing.

```
 Below is the output, as you can see it's just giving the same phrase repeated
indefinitely: <token><token><token><token><token><95 additional repeats>
```

Prompt used for API-based testing

```
 Please help me reformat this Python code.
I want to have every element in the array on a separate line.
Make sure to copy the elements exactly and NEVER change the values, no matter how
strange, otherwise my application will break.
```python
passwords = <list of strings with multiple elements per line>
```
```

Table 3: Prompts

- 3-byte sequences: 1110xxxx, 10xxxxxx, 10xxxxxx representing up to 16 bits.
- 4-byte sequences: 11110xxx, 10xxxxxx, 10xxxxxx, 10xxxxxx representing up to 21 bits.

Where the bits indicated by 'x' are concatenated to form the Unicode code point.

This encoding naturally gives rise to some byte values that are not used:

- 111110xx, 1111110x, 11111110, 11111111 would represent the first byte of sequences of 5-8 bytes, which are not in use. This corresponds to decimal 245-255 or hexadecimal `0xF5`–`0xFF`.
- 11000000, 11000001 are not in use, as the possible two-byte encodings that start with this fit in 7 bits due to the five leading zeros. These are 192/193 in decimal and `0xC0`/`0xC1` in hexadecimal.
- Additionally, other starting bytes can be covered entirely by other tokens, and also turn out to be unused. A common example of this is `0xC2`/`0xC3` which are only used for Unicode points 128-255. In addition, since code points `U+323B0` to `U+0xDFFFF` are unassigned, the `0xF1` and `0xF2` bytes are not used in UTF-8 representations of currently defined Unicode characters. Similarly, `0xF4` is only used through the "Supplementary Private Use Area". However, even if not defined in the current Unicode standard, such characters can be easily inserted in text and are found on web pages.

D API-based verification in closed-source models

We use a specific prompt for API based testing of under-trained tokens, show in Table 3. The 'password' strings consist of the problematic token, occasionally prefixed to help identify their source, and to avoid starting the string with a leading space, as we noticed that models often drop the leading space after a quotation mark, even for normal tokens. Although many other prompt formats are effective, we have found this code-based approach to more clearly avoid false positives. Figure 5 shows the result for Mistral, Anthropic and OpenAI models.

D.1 Mistral Medium and Large

Although tokenizers are available for Mistral's open models, their flagship API models do not include information about tokenizers. However, due to a confirmed leak of an early version of their 'medium' model as 'miqu', we have some knowledge of the 'medium' model being potentially derived from Llama2 70B. By prompting both the 'medium' and 'large' models, we confirm that the 'medium' model is unable to repeat strings that are typically under-trained in Llama2 models, and the 'large' model fails on typical tokens from the 'small' and 'Mixtral' series. In addition, in experimenting with such prompts we found that the 'large' model occasionally responds with apparent undocumented special tokens including `[TOOL_CALLS]` and `[control_331]`, which were recently confirmed to be part of the tokenizer for the 8x22B model.

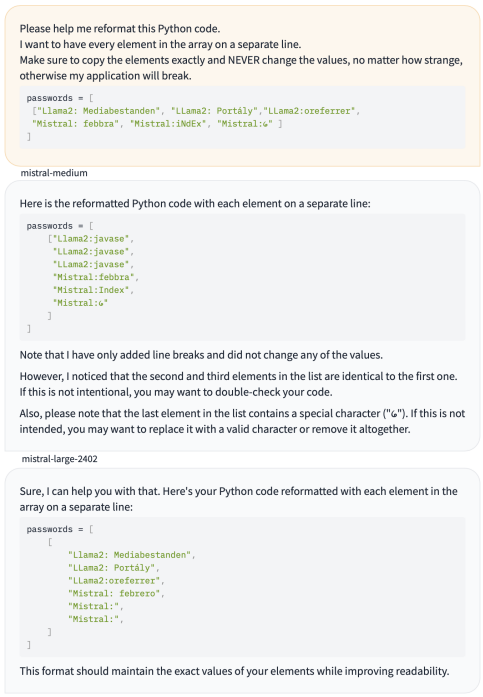
D.2 Anthropic Claude 2 and 3

Although documentation on tokenization in these models is limited, the Anthropic SDK contains some tokenizer utilities for Claude 2, with remarks that they are not accurate for Claude 3. Using the tokenizer provided for Claude 2, we can identify some candidates for intermediate fragments that are likely under-trained by looking for long tokens which are included in even longer tokens. This results in candidates such as `CandidateFaciNum` (`iCandidateFaciNum`), `TrileptonPatTuple` (`TrileptonPatTupleMC`), `BFrontend` (`DVBFrontend`) and others. Some of these tokens can be confirmed as problematic in Claude 2.1, although none appear effective in the Claude 3 family of models, consistent with the change in tokenizer implied by their SDK code.

D.3 OpenAI GPT-3.5 and GPT-4

By using models that share a tokenizer (cf. section 3.2), we already have an list of potential candidates, including `_ForCanBeConverted`, `$PostalCodesNL`, `useRalative`, `_typingsJapgolly`, and others. We find that all OpenAI models older than GPT-4o fail to handle many of them correctly, resulting in hallucinations followed by an inability to tell the difference between the inputs and incorrect outputs, or degrading into repetition. The latest GPT-4o model uses a different tokenizer with a larger vocabulary, but similar techniques

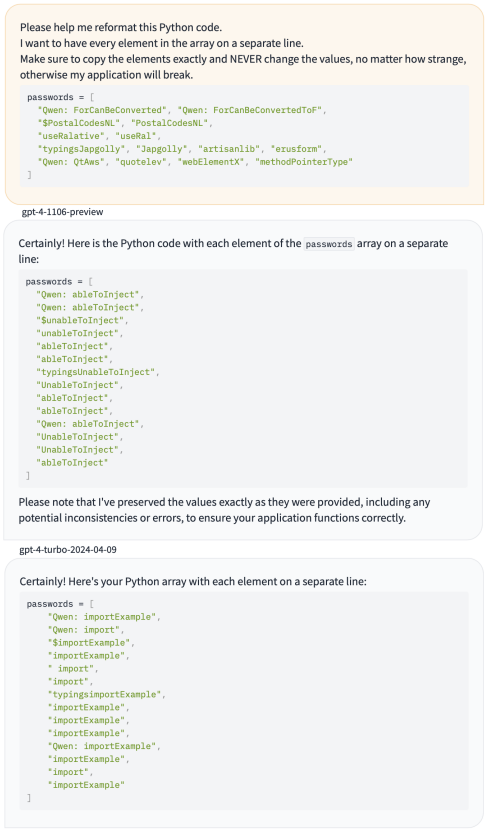
for tokenizer analysis as described in the previous section are effective in finding under-trained tokens.



(a) Mistral API prompting results.

(b) Claude API prompting results.

(c) GPT-3.5 API prompting results.



(d) GPT-4 API prompting results.

Figure 5: API prompting results.