From Bytes to Subwords: Challenges of Input Representations in NLP

Rob van der Goot
IT University of Copenhagen
robv@itu.dk

Abstract

A first decision for any automated natural language processing system is the granularity of the input units. Traditionally, characters or words have been used, but recently, subwords have become the standard. In this paper, we investigate trends in input processing steps and discuss common shortcomings in this foundational first step of model design. We start by providing an overview of currently used tokenizers, showing that there is only minimal variety, with three highly similar designs dominating current models, and many of the tokenizers being exact duplicates. Next, we reconsider Unicode normalization strategies. Previous work has recommended applying consistent normalization; however, we argue that this removes signal and we show how this can harm performance for language classification. Finally, we take a closer look at UTF-8 character encoding, the very first layer of representation used in many language models. We argue that UTF-8 is not optimized for efficiency, nor for fairness across languages, and propose proof of concept alternatives focused on fairness and efficiency. Based on our findings, we recommend future work to 1) put more thought into subword segmentation and explore more diversity, 2) apply normalization only when beneficial 3) consider alternative character encodings for models operating on the byte-level.[1]

1 Input Encodings in NLP

For building models for the automatic processing of written natural language, one of the first decisions to make is how to represent the input. In this section, we review common input processing steps that are used in Natural Language Processing (NLP) models; we start with the smallest units (bytes, characters), and then look at subword segmentation strategies.

Written natural language is commonly expressed as sequences of characters. To store texts, character encodings are used to represent characters as bytes. UTF-8 is the most commonly used character encoding today, it is used on 98.3% of the websites indexed by W3C.[2] To the best of our knowledge, all text models on HuggingFace use UTF-8.[3] It should be noted that there is a stream of work on vision-based language models that render text as an image, and use the pixels as an input (e.g. Rust et al., 2022). These models are not dependent on UTF-8, and do not suffer from the same problems. For our main analyses, we consider these models out-of-scope, but we include a discussion of the relevance of our results for vision-based language models in Section 5.

Since the 159,866 characters currently included in Unicode do not fit into a single byte, UTF-8 has a flexible byte-length (1-4). For backwards compatibility to ASCII, it represents the main Latin characters, numbers, and control characters in the first byte, but also an array of continuation bytes, which are a prefix for the higher-indexed characters. To clarify how this leads to differences in representations of text, consider the following examples:

b	ø	r	n
62	C3 B8	72	6E
д	і	т	и
1076	1110	1090	1080
D0 B4	D1 96	D1 82	D0 B8
孩	子	们	
23401	23376	20204	
E5 AD A9	E5 AD 90	E4 BB AC	

Figure 1: Character representations for the word "children" in Danish, Ukranian, and Chinese.

These examples clearly show that different lan-

[1] Code available on: https://bitbucket.org/robvanderg/inputencodings/

[2] https://w3techs.com/technologies/cross/character_encoding/ranking

[3] This was confirmed by searching for the encoding flag in the transformers models classes.

Findings of the Association for Computational Linguistics: ACL 2026, pages 10911–10919

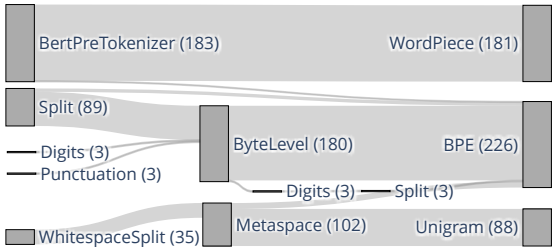

Figure 2: Frequency of sequences of tokenization steps for 500 most downloaded models. To be read from left to right, e.g. there are 193 tokenizers using BertPreTokenizer, of which 181 are followed by a WordPiece segmentation, and 2 by BPE.

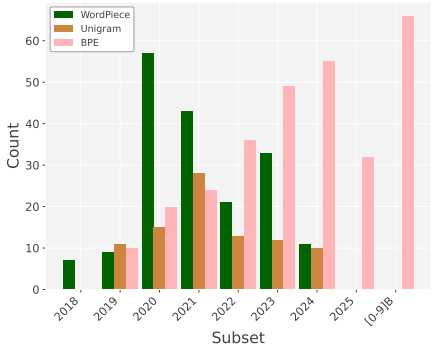

Figure 3: Counts of subword types for subsets of the top 500 most downloaded models. We count per year as well as for models containing the regex "$[0-9]$B".

guage are treated differently, while Latin characters are represented by a single byte, many other characters need 2-3 bytes. Unicode characters are stored in 'blocks', which are subsets of scripts that are stored in continuous regions, this is reflected in the examples by the shared prefix bytes.

While there is a stream of models that use bytes or characters directly as input (see for an overview Mielke et al. (2021)), using these as input has some downsides; they lead to long sequences, and in many writing systems characters have no connection to meaning. Words, on the other hand, lead to a very large vocabulary, less overlap, and their segmentations are non-trivial to obtain. Hence, most modern models make use of subwords as their input. Subwords are sequences of characters that commonly co-occur, and their segmentations are usually learned through unsupervised algorithms (e.g. Gage, 1994; Schuster and Nakajima, 2012; Kudo and Richardson, 2018).

Besides the subword algorithm, there is a wide variety of possible pre-processing steps. The combination of pre-processing steps included in a model is often referred to as its pre-tokenizer. Decisions to make include: the type of smallest input units used (bytes versus characters), whitespace handling, punctuation separation, and types of normalization applied. All these design decisions affect the performance of the resulting models (e.g. Rust et al., 2021; Dagan et al., 2024; Reddy et al., 2025; Raj S et al., 2025) as well as costs (Ahia et al., 2023; Lundin et al., 2025).

To map the current landscape of pre-tokenization and subword segmentation in language models, we collect the tokenizers of the 500 most downloaded models from Huggingface,[4] and extract the sequence of Huggingface classes used in the tokenizer, including the sequence of pre-tokenizers and the final subword segmentation algorithm.[5] From the resulting tokenization pipelines (Figure 2), we see that the main subword segmentation algorithms (BPE, Wordpiece, and Unigram) are all frequently used, and each of them has 1-2 prominent pre-tokenizer designs. Furthermore, we can see that byte-level inputs are commonly used.

When breaking down the tokenizers by their Huggingface creation date (Figure 3)[6] we see clear trends. BPE is steadily increasing, whereas wordpiece is decreasing. Unigram usage is generally more stable, but has a small peak in creation dates around 2021. Finally, we look at the subset of models that contain the regular expression "$[0-9]$B" in their name, which is an indication that they are recent generative language model with at least 1B weights. 100% of these models is using a BPE tokenizer (Figure 3).

In this opinion paper, we will analyze three different design decisions commonly used for input processing in NLP models. Based on our observations, we recommend model builders to consider:

- exploring more variety in tokenizer design, including language-specific tokenizer design, pre-tokenizer steps, segmentation algorithms, and other hyperparameters.
- making usecase based decisions for normalization, or ensure that models are robust by normalizing only part of the training data.

[4] As of 27-11-2025. This list is not a perfect reflection of usage, as some models may rank higher e.g. because they are the defaults in toolkits or used in tutorials. Nevertheless, we believe this sample will reflect the main trends.

[5] The classes are described on: huggingface.co/docs/tokenizers/api/pre-tokenizers

[6] This is a proxy to creation date/age; not all models are directly included on Huggingface upon release

- designing character encodings that align with desiderata of the downstream model.

2 Lack of Diversity

Dagan et al. (2024) show (in their Appendix C) that for coding-based models, tokenizers are often directly re-used from older models. In this section we discuss tokenizer re-use for text-based models more generally. The frequencies in Figure 2 show that there are five main tokenizer pipelines designs (>=35 freq.); WordPiece has one common pre-tokenization sequence, while BPE and Unigram have two frequent pre-tokenization sequences (Split or no Split, and WhitespaceSplit or not respectively). However, when inspecting the differences in the two main pre-processing decisions of BPE and Unigram, we found that they do not lead to substantially different behavior. First, the only effect of WhitespaceSplit before MetaSpace is that it unifies white space characters that are not the standard spaces (tabs, newlines, etc.).

The second main pre-tokenization difference is that 89/180 of the ByteLevel-BPE use the Split class (which split texts based on a regular expression). Furthermore, we find that in all 91 tokenizers with the ByteLevel-BPE sequence, a similar split is defined in the code of the tokenizer class. When inspecting the regular expressions that are used, we find that they are highly similar; all of them specify a separation based on sequences of whitespaces, numbers, and punctuation. The vast majority also separates English contractions ('m, 'll, etc.). We report more details on this in Appendix A.

Explicit separation of English contractions is not only an indication of an English-centric bias, it is also an odd design decision, as all regular expressions that include it also define a split for punctuation characters. This means that specifying contractions does not lead to different behavior for contractions, but since they occur earlier, they do affect other occurrences of the same characters (e.g. "e'mail" will be split into "e 'm ail" because 'm is included). This suggests that for most tokenizers, the regular expression is simply copied from other models without giving their design much thought.

Since for both BPE and Unigram the pre-tokenization pipelines are in fact highly similar across all instances, we conclude that there are three main different tokenizer pipeline architectures in use, which is in stark contrast compared to the linguistic variety present in languages across the world. Besides the architecture, other design decisions also have an impact on performance, such as vocabulary size (Reddy et al., 2025), and training data (Dagan et al., 2024; Rust et al., 2021). However, within our top-500 sample, there are only 255 unique tokenizers, confirming that many language models use exact copies of other popular models.

Recommendation From a broader perspective, it has been shown that different types of languages and scripts have preferences for different tokenizer designs (Bostrom and Durrett, 2020; Limisiewicz et al., 2023; Hou et al., 2023; Wegmann et al., 2025; Reddy et al., 2025). Whereas many models in our top-500 sample support multiple languages, they all use a single tokenizer design for all of their input, although some of them are trained separately on languages and merged afterwards (Imani et al., 2023; Liang et al., 2023), their tokenization architecture is exactly the same. The decision to use the same architectural design for each language and writing system is probably because of implementation convenience and to limit the number of hyperparameters to tune, but it likely leads so suboptimal segmentations for many languages.

3 Normalization is Lossy

There are four official Unicode normalization standards; they perform a variety of normalization replacements, including 1) converting accents into a standard form (either composed or decomposed) 2) correcting the order of sequences of diacritics. 3) whitespace standardization 4) mapping of single characters (e.g. some numbers are normalized: $^2 \mapsto 2$, or equivalent-looking characters are unified). However, it is non-trivial to detect which normalization is used when it is combined with other pre-processing steps, and without having the exact training code and commands. As a first inspection, we count the different types of normalization across HuggingFace model types: NFC: 22, NFD: 21, NFKC: 1, NFKD: 4.[7] A closer inspection shows that NFC is often directly integrated (i.e. not optional) into the model code, whereas the other normalizations are optional (i.e., a hyperparameter). To get a better overview of what is used in popular models, we also inspect the normalization included in the top-500 model pipelines (details in Appendix B). Results from this investigation confirm that NFC (242 versus 9 for the others) is the most common normalization.

[7] Note that these per model class, not specific models.

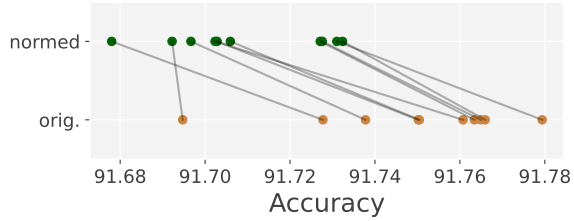

Figure 4: Performance (accuracy) of language classification when using non-normalized versus normalized data. Each dot represents the accuracy of a model trained with a specific seed.

Any applied normalization leads to lossy tokenization, violating the decode-encode test: (`word == decode(encode(word))`) (Kudo and Richardson, 2018; Dagan et al., 2024; Jang et al., 2025). Lossless tokenization has potential benefits because the original signal is fully kept and alignment of subwords to inputs is easier; at the same time, it can also lead to treating two similar-looking characters as completely different data points. Gorman and Pinter (2025) indeed show that using consistent normalization leads to small, but cheap performance gains for parsing of a Hindi treebank. However, in some cases differences in encodings are not an arbitrary signal; they are a result of an input method, and different people will thus use different character encodings for the same character.

To evaluate the effect of this, we perform a case study on the language classification task. We use the setup from van der Goot (2025), because of their wide language coverage, and models are trained from scratch. We use their mixed-domain, 2,307 languages, 1,000 instances per language setup. We use their implementation of their best performing model, which is a Naive Bayes classifier with 1-5 character n-grams binary features. We add an optional NFC normalization step, because it is the most commonly used normalization in language models. We generate 10 different data splits with different seeds.

Results are shown in Figure 4. Because some seeds might lead to more challenging data, we connect the runs that use the same seed with a line, showing that in all 10 cases the non-normalized version performs better. The NFC model obtains 91.70 average accuracy, and the non-normalized model obtains 91.75. When applying an ASO test (Del Barrio et al., 2018; Dror et al., 2019; Ulmer et al., 2022) with a confidence level $\alpha = 0.05$, the score distribution of the non-normalized is stochastically dominant over the normalized version ($\epsilon_{min} = 0.01$, < 0.5 indicates significance in ASO tests). Although the gains are small , they are consistent, significant, and cheap to obtain.

Recommendation Athough our results show improved performance when not normalizing, there will definitely be situations where normalization is beneficial (e.g. Gorman and Pinter, 2025), at the same time, if tokenization should be lossless at inference time, but the models should be robust against non-normalized input and pass the decode-encode test, one could normalize only a portion of the data, so that the model becomes robust against different representations of the same character.

4 UTF-8 is neither Fair nor Efficient

While UTF-8 has many useful properties, it has been designed with desiderata that do not necessarily overlap with requirements for NLP models. The main focus is on coverage and backwards compatibility with itself as well as ASCII. The single-byte positions (i.e., the most efficient) of UTF-8 are mostly assigned to characters from the Latin script and control characters, many of which are remnants of legacy control characters dating back to typewriters. Furthermore, due to the sequential updates, newer additions automatically have a higher chance of being placed in the 3-4 byte ranges, and scripts are not guaranteed to be sequential.

Backwards compatibility of character encodings is less relevant when designing NLP models. Instead, desiderata include: coverage, efficiency, fairness (across scripts/languages), relevant overlap for multi-byte characters. The last three of these are not main focus points in the current design of UTF-8. How heavy each requirement should weigh in the design should depend on the use cases of the final model. For example, for monolingual models, fairness across scripts is easier to obtain, as the number of scripts is smaller.

Previous work has already shown that having invalid (Jang et al., 2025; Gorman and Pinter, 2025) or even rare (Land and Bartolo, 2024) UTF-8 in the input leads to performance issues, and also that byte-level models are prone to output invalid UTF-8 (Firestone et al., 2025). However, to the best of our knowledge there is limited work in designing more fair or efficient character encodings for NLP purposes. Moryossef et al. (2025) propose to remap some of the control bytes input into language models' special tokens. Moon et al. (2025)

propose a remapping of multi-byte characters that reduces bit-redundancy and leads to shorter inputs. The most rigorous alternative encoding is proposed by Land and Arnett (2025), who design a new encoding based on Unicode script and category information.[8] In the remainder of this section, we will propose three proof-of-concept strategies to achieve some of the aforementioned desiderata. As previously mentioned, depending on use-cases, a combination of these approaches should be considered, with focus points on the desiderata that are most important for the intended setup.

Efficiency For a case study, we sample the first 10,000 characters for the 1,742 language-script combinations of Fineweb2 (Penedo et al., 2025). The resulting total data collection occupies 34,126,685 bytes when encoded with UTF-8. To reduce this, we can simply sort the characters by frequency (we apply add-1 smoothing to include all characters), then store the most frequent characters in the first byte, the following in 2-byte positions, and the least common characters in 3 bytes. Variables to decide here are the number of continuation bytes for the 2-byte positions and the 3-byte positions. We optimized these (Appendix C), and are able to store the exact same data in only 30,479,531 bytes, saving ˜10% through a simple remapping.

Fairness Our next approach focuses on overlap. There are 175 scripts in Unicode 17, so we reserve the first byte for defining the script. The interpreter needs to know for each script whether it can be represented in a single byte or in more bytes. 150 scripts fit in a single byte, 24 scripts need two bytes, and only the Han script requires three bytes to represent all characters. It should be noted that the clusters could be refined, e.g., by merging scripts with the common class, using sub-ranges of scripts, or having fully data-driven clusters. This clustering approach can also be used to shorten sequences by having script-change bytes, then the prefix is only required once for sequential in-script characters, similar as proposed by Moon et al. (2025).

Unseen characters Another problem is that even though byte-level representations can represent any UTF-8 character, if a character is never or rarely seen during training, it will not have a high-quality representation. There are whole ranges in Unicode that are likely not seen during pretraining of most models. In fact, our relatively diverse text sample with the first 10,000 characters of 1,742 languages of Fineweb2 only contains 8,739/159,866 Unicode characters (˜5.5%). If a character is not in the training data, but other characters from the same script are, it still makes sense to have some overlap in the byte representations. This is indeed happening in most cases; however, the last byte specifying the exact position is not going to be informative, as it only overlaps with arbitrary other characters and leads to an unseen sequence. In other words, even if all bytes are seen before in byte-level models, the unseen character problem is not solved in byte-level models, as their representation will not be representative. Hence, it could make sense to reserve an index for each script for unknown tokens, or perhaps even sub-categories of unknown characters, similar to how unknown words used to be processed (e.g. Klein and Manning, 2003).

[8] This is a more thorough implementation of our "Fairness" strategy, we unfortunately only found their paper after ours was submitted.

5 Discussion: Vision-based Alternatives

Vision based language models (e.g. Salesky et al., 2021; Rust et al., 2022; Kesen et al., 2025) solve many of the issues presented in this paper. There is no pre-tokenizer or subword algorithm, so the findings of Section 2 do not hold. However, there are other decisions to be made in the pre-processing design, for example the window size and overlap. It is easy to imagine that different languages have different optimal values here as well, so similarly as for text-based language models, diversity should be considered. The normalization issues presented in Section 3 are not relevant for vision based models, as there is no need for a normalization step. Finally, vision-based models are also not vulnerable to biases in the byte-sequences of UTF-8 (Section 4).

6 Conclusion

In this paper, we have given an overview of common approaches used in the input representations of current language models and inspected potential weaknesses. Based on our findings, we argue that: 1) Current tokenization design is homogeneous, and contains suboptimal remnants 2) character normalization should be used with care 3) UTF-8 is far from optimal for NLP use-cases, and when using models that rely on it, we should consider designing alternative encodings with specific desiderata in mind. We hope this position paper leads to more thoughtful design and more diversity of tokenizers.

Acknowledgements

This paper has greatly benefited from two groups of people: the MaiNLP research group and 2/3 of the reviewers. I would like to thank them for all their thoughtful and concrete comments.

Limitations

For collecting statistics, we mainly focus on the top-500 downloaded models from Huggingface. This is not only a snapshot, but also does not reflect which models are most frequently used, as the number of downloads is distorted by models that are included in tutorials or toolkit defaults. Furthermore, it does not incorporate derivative/finetuned models (e.g. bert-base-cased and bert-base-cased finetuned on a specific dataset are counted separately).

Besides the data-driven approaches recommended in Section 4, defining a character encoding based on knowledge of the language/script (e.g. Ansary et al., 2024) is a better alternative, but requires a careful design by experts for each script. Furthermore, downstream evaluations are necessary to empirically confirm that our proposed solutions in Section 4 are beneficial, however, it is computationally expensive to evaluate solutions that target multiple scripts, as a lot of data and compute is required to train models of a relevant scale.

There are many additional design decisions that have an impact on the final tokenizer that we did not discuss in detail because they have been covered in previous work. For example, the size of the training data (Reddy et al., 2025), word-initial versus word-final continuation marking (Jacobs and Pinter, 2022), or inference methods (Uzan et al., 2024).

References

Orevaoghene Ahia, Sachin Kumar, Hila Gonen, Jungo Kasai, David Mortensen, Noah Smith, and Yulia Tsvetkov. 2023. Do all languages cost the same? tokenization in the era of commercial language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 9904–9923, Singapore. Association for Computational Linguistics.

Nazmuddoha Ansary, Quazi Adibur Rahman Adib, Tahsin Reasat, Asif Shahriyar Sushmit, Ahmed Imtiaz Humayun, Sazia Mehnaz, Kanij Fatema, Mohammad Mamun Or Rashid, and Farig Sadeque. 2024. Unicode normalization and grapheme parsing of Indic languages. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 17019–17030, Torino, Italia. ELRA and ICCL.

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.

Gautier Dagan, Gabriel Synnaeve, and Baptiste Rozière. 2024. Getting the most out of your tokenizer for pre-training and domain adaptation. In *Proceedings of the 41st International Conference on Machine Learning*, pages 9784–9805.

Eustasio Del Barrio, Juan A Cuesta-Albertos, and Carlos Matrán. 2018. An optimal transportation approach for assessing Almost Stochastic Order. In *The Mathematics of the Uncertain*, pages 33–44. Springer.

Rotem Dror, Segev Shlomov, and Roi Reichart. 2019. Deep dominance - how to properly compare deep neural models. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 2773–2785. Association for Computational Linguistics.

Preston Firestone, Shubham Ugare, Gagandeep Singh, and Sasa Misailovic. 2025. UTF-8 plumbing: Byte-level tokenizers unavoidably enable LLMs to generate ill-formed UTF-8. In *Second Conference on Language Modeling*.

Philip Gage. 1994. A new algorithm for data compression. *The C Users Journal*, 12(2):23–38.

Kyle Gorman and Yuval Pinter. 2025. Don't touch my diacritics. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 2: Short Papers)*, pages 285–291, Albuquerque, New Mexico. Association for Computational Linguistics.

Jue Hou, Anisia Katinskaia, Anh-Duc Vu, and Roman Yangarber. 2023. Effects of sub-word segmentation on performance of transformer language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7413–7425, Singapore. Association for Computational Linguistics.

Ayyoob Imani, Peiqin Lin, Amir Hossein Kargaran, Silvia Severini, Masoud Jalili Sabet, Nora Kassner, Chunlan Ma, Helmut Schmid, André Martins, François Yvon, and Hinrich Schütze. 2023. Glot500: Scaling multilingual corpora and language models to 500 languages. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1082–1117, Toronto, Canada. Association for Computational Linguistics.

Cassandra L Jacobs and Yuval Pinter. 2022. Lost in space marking. *arXiv preprint arXiv:2208.01561*.

Eugene Jang, Kimin Lee, Jin-Woo Chung, Keuntae Park, and Seungwon Shin. 2025. Improbable bigrams expose vulnerabilities of incomplete tokens in byte-level tokenizers. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 18220–18227, Suzhou, China. Association for Computational Linguistics.

Ilker Kesen, Jonas F. Lotz, Ingo Ziegler, Phillip Rust, and Desmond Elliott. 2025. Multilingual pretraining for pixel language models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 29594–29611, Suzhou, China. Association for Computational Linguistics.

Dan Klein and Christopher D. Manning. 2003. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, pages 423–430, Sapporo, Japan. Association for Computational Linguistics.

Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, Brussels, Belgium. Association for Computational Linguistics.

Sander Land and Catherine Arnett. 2025. BPE stays on SCRIPT: Structured encoding for robust multilingual pretokenization. In *TokShop*.

Sander Land and Max Bartolo. 2024. Fishing for Magikarp: Automatically detecting under-trained tokens in large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 11631–11646, Miami, Florida, USA. Association for Computational Linguistics.

Davis Liang, Hila Gonen, Yuning Mao, Rui Hou, Naman Goyal, Marjan Ghazvininejad, Luke Zettlemoyer, and Madian Khabsa. 2023. XLM-V: Overcoming the vocabulary bottleneck in multilingual masked language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13142–13152, Singapore. Association for Computational Linguistics.

Tomasz Limisiewicz, Jiří Balhar, and David Mareček. 2023. Tokenization impacts multilingual language modeling: Assessing vocabulary allocation and overlap across languages. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 5661–5681, Toronto, Canada. Association for Computational Linguistics.

Jessica M Lundin, Ada Zhang, Nihal Karim, Hamza Louzan, Victor Wei, David Adelani, and Cody Carroll. 2025. The token tax: Systematic bias in multilingual tokenization. *arXiv preprint arXiv:2509.05486*.

Sabrina J Mielke, Zaid Alyafeai, Elizabeth Salesky, Colin Raffel, Manan Dey, Matthias Gallé, Arun Raja, Chenglei Si, Wilson Y Lee, Benoît Sagot, and 1 others. 2021. Between words and characters: A brief history of open-vocabulary modeling and tokenization in NLP. *arXiv preprint arXiv:2112.10508*.

Sangwhan Moon, Tatsuya Hiraoka, and Naoaki Okazaki. 2025. Bit-level bpe: Below the byte boundary. *arXiv preprint arXiv:2506.07541*.

Amit Moryossef, Clara Meister, Pavel Stepachev, and Desmond Elliott. 2025. Back to bytes: Revisiting tokenization through UTF-8. *arXiv preprint arXiv:2510.16987*.

Guilherme Penedo, Hynek Kydlíček, Vinko Sabolčec, Bettina Messmer, Negar Foroutan, Amir Hossein Kargaran, Colin Raffel, Martin Jaggi, Leandro Von Werra, and Thomas Wolf. 2025. FineWeb2: One pipeline to scale them all–adapting pre-training data processing to every language. *arXiv preprint arXiv:2506.20920*.

Bharath Raj S, Garvit Suri, Vikrant Dewangan, and Raghav Sonavane. 2025. When every token counts: Optimal segmentation for low-resource language models. In *Proceedings of the First Workshop on Language Models for Low-Resource Languages*, pages 294–308, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Varshini Reddy, Craig W Schmidt, Yuval Pinter, and Chris Tanner. 2025. How much is enough? the diminishing returns of tokenization training data. In *Proceedings of the 42 nd International Conference on Machine Learning*. PMLR.

Phillip Rust, Jonas F Lotz, Emanuele Bugliarello, Elizabeth Salesky, Miryam de Lhoneux, and Desmond Elliott. 2022. Language modelling with pixels. In *The Eleventh International Conference on Learning Representations*.

Phillip Rust, Jonas Pfeiffer, Ivan Vulić, Sebastian Ruder, and Iryna Gurevych. 2021. How good is your tokenizer? On the monolingual performance of multilingual language models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 3118–3135, Online. Association for Computational Linguistics.

Elizabeth Salesky, David Etter, and Matt Post. 2021. Robust open-vocabulary translation from visual text representations. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 7235–7252, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Mike Schuster and Kaisuke Nakajima. 2012. Japanese and Korean voice search. In *2012 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 5149–5152. IEEE.

Dennis Ulmer, Christian Hardmeier, and Jes Frellsen. 2022. deep-significance: Easy and meaningful signifcance testing in the age of neural networks. In *ML Evaluation Standards Workshop at the Tenth International Conference on Learning Representations*.

Omri Uzan, Craig W. Schmidt, Chris Tanner, and Yuval Pinter. 2024. Greed is all you need: An evaluation of tokenizer inference methods. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 813–822, Bangkok, Thailand. Association for Computational Linguistics.

Rob van der Goot. 2025. Identifying open challenges in language identification. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 18207–18227, Vienna, Austria. Association for Computational Linguistics.

Anna Wegmann, Dong Nguyen, and David Jurgens. 2025. Tokenization is sensitive to language variation. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 10958–10983, Vienna, Austria. Association for Computational Linguistics.

Appendix

A Regular Expression Split Detection

To obtain an overview of the strategies used for pre-splitting tokens, we inspect the regular expressions used. We focus again on the 500 most frequently downloaded models. We first obtain the regular expressions of all 89 models that define the Split class. Next, we inspect the byte-level BPE tokenizers that do not have the Split class included in their definition and find that these 91 tokenizers are from 11 different tokenizer classes. After inspecting their code, we find that they all define a regular expression in their code, and they are all equivalent (first row in Figure 5). Combining both the Split class and the integrated regular expressions we obtain the frequencies reported in Figure 5.

B Normalization detection

For detecting Unicode normalization on the class level, we recursively search for all occurences of the unicodedata library in the models directory of the transformers library. This will give us estimates for model classes, however, it excludes cases where normalization is happening through another function, and it does not take frequenty of model use into account. To obtain a clearer view of commonly used normalizations we perform an additional empirical evaluation on the top-500 most downloaded models of Huggingface, which is described in the following paragraph.

To automatically detect which normalization is used by which model, we first inspect the `tokzr.backend_tokenizer.normalizer` variable where the normalizer class can be defined. As a second, more robust step, we investigate the behaviour of the tokenizer. It could be that the normalizer is defined in external code, or even as a processor in the tokenizer files. Hence, we create a list of characters that are prone to normalization, and evaluate whether the output of all examples for a certain tokenizer (after encoding and decoding) matches exactly with the output of a Unicode normalization standard. We only count cases where all of the test characters match (some models remove diacritics, and are thus not matched by our approach), and take the union of both detection methods as final set of normalization applied by a certain tokenizer.

C Remapping characters to indices based on frequency

We first count all characters in the dataset, and add 1 to their counts to ensure each character can be represented. We then sort the characters based on their frequency. We store the first n characters in the first byte, where n = 256 - total number of continuation bytes We define two types of continuation indices, one for 2-byte characters (x), and one for 3 byte characters (y). We use continuation bytes instead of bits to use the storage more efficiently. This makes the number of actual characters that can be be stored in the first byte: $256 - x - y$. We have $x * 256$ 2-byte characters and y*256*256 3-byte characters.

Based on this, we use a script that compares all options of x and y with the real character counts from our FineWeb2 sample, leading to an optimal of x = 16 and y=3. It should be noted that when applying such an encoding, it would be beneficial to create the mapping so that characters from the same script have more overlap in bytes, instead of a purely frequency based mapping. It should also be noted that further optimization could be done when using different lengths of bits, e.g. (Moon et al., 2025) use 6 bits prefixes.

```
91  's|'t|'re|'ve|'m|'ll|'d| ?\p{L}+| ?\p{N}+| ?[^\s\p{L}\p{N}]+|\s+(?!\S)|\s+
38  (?i:'s|'t|'re|'ve|'m|'ll|'d)|[^\r\n\p{L}\p{N}]?\p{L}+|\p{N}| ?[^\s\p{L}\p{N}]+[\r\n]*|\s*[\r\n]+|...
17  (?i:'s|'t|'re|'ve|'m|'ll|'d)|[^\r\n\p{L}\p{N}]?\p{L}+|\p{N}{1,3}| ?[^\s\p{L}\p{N}]+[\r\n]*|\s*[\r...
 9  's|'t|'re|'ve|'m|'ll|'d|[\p{L}]+|[\p{N}]|[^\s\p{L}\p{N}]+
 7  <\|startoftext\|>|<\|endoftext\|>|'s|'t|'re|'ve|'m|'ll|'d|[\p{L}]+|[\p{N}]|[^\s\p{L}\p{N}]+
 4  \p{N}{1,3}
 2  [^\r\n\p{L}\p{N}]?[\p{Lu}\p{Lt}\p{Lm}\p{Lo}\p{M}]*[\p{Ll}\p{Lm}\p{Lo}\p{M}]+|[^\r\n\p{L}\p{N}]?[\...
 2  [^\r\n\p{L}\p{N}]?[\p{Lu}\p{Lt}\p{Lm}\p{Lo}\p{M}]*[\p{Ll}\p{Lm}\p{Lo}\p{M}]+(?i:'s|'t|'re|'ve|'m|...
 2  ?[^(\s|[.,!?. . . ])]+
 1  (\[[^\]]+]|Br?|Cl?|N|O|S|P|F|I|b|c|n|o|s|p|\(|\)|\.|=|#|-|\+|\\|\/|:|~|@|\?|>|\*|\$|\%[0-9]{2}|[0...
```

Figure 5: Frequency counts of regular expressions used to split tokens (first 100 characters shown).