Splintering Nonconcatenative Languages for Better Tokenization

Anonymous ACL submission

Abstract

Common subword tokenization algorithms like BPE and UnigramLM assume that text can be split into meaningful units by concatenative measures alone. This is not true for languages such as Hebrew and Arabic, where morphology is encoded in root-template patterns, or Malay and Georgian, where split affixes are common. We present SPLINTER, a pre-processing step which rearranges text into a linear form that better represents such nonconcatenative morphologies, enabling meaningful contiguous segments to be found by the tokenizer. We demonstrate SPLINTER's merit using both intrinsic measures evaluating token vocabularies in Hebrew, Arabic, and Malay; as well as on downstream tasks using BERT-architecture models trained for Hebrew.

1 Introduction

Large language models (LLMs) have become pivotal in natural language processing (NLP), offering extensive utility across diverse applications. Central to constructing an LLM is producing basic input units from the text sequence, for which subword tokenization is still the standard approach, using methods such as byte-pair encoding (BPE) (Sennrich et al., 2016), WordPiece (Schuster and Nakajima, 2012), and UnigramLM (Kudo, 2018). However, subword tokenizers exhibit diminished effectiveness in nonconcatenative languages (NCLs) such as Hebrew and Arabic (Klein and Tsarfaty, 2020). While tokenizers assume linear segmentation of words, NCLs' units of meaning are typically intertwined within words, and **cannot be separated linearly**, as the root letters are not adjacent (Khaliq and Carroll, 2013). A Hebrew example is provided in Table 1. in the Hebrew word לעבוד 'to work', the root letters are 'ע', 'ב', and 'ד', placed in an infinitive morphological template manifested by the locations of 'ל' and 'ו'. This characteristic forces linear tokenizers to split words into morphologically-incoherent tokens, losing the downstream models' ability to generalize across various forms of the same lemma, and eventually reducing model performance when applied to a large variety of tasks such as text generation and translation (Keren et al., 2022; Levi and Tsarfaty, 2024; Shmidman et al., 2024).

(a)	עבדתי על המצגת. 'I worked on the presentation.'
(b)	העבודה הקשה השתלמה. 'The hard work paid off.'

Table 1: Examples of Hebrew text (read from right to left) exemplifying its nonconcatenative morphology. The root **עבד** 'work' appears in both sentences, but in (a) it comprises a linear segment of the text whereas in (b) it is broken by the templatic character ו.

We present SPLINTER, a statistical algorithm for *linearizing* NCL text through rearranging the text sequence by iteratively pruning characters from words, with the intent of isolating characters representing template forms. The manipulated text can then be input into any ordinary linear tokenizer for processing as usual, adapting the NCL data into the morphologically-concatenative input BPE and its like expect. We show that vocabularies and models trained over SPLINTER-processed text outperform those starting from raw NCL text on both intrinsic and extrinsic measures in Hebrew, Arabic, and Malay.[1]

2 Tokenizing with Splinters

When designing our approach, our goal was to create a relinearized sequence for words in NCL languages, while adhering to several constraints. One constraint is that the transformation must be reversible, ensuring that the new representation can always be converted back to the original text. In

[1]We will release our code upon publication.

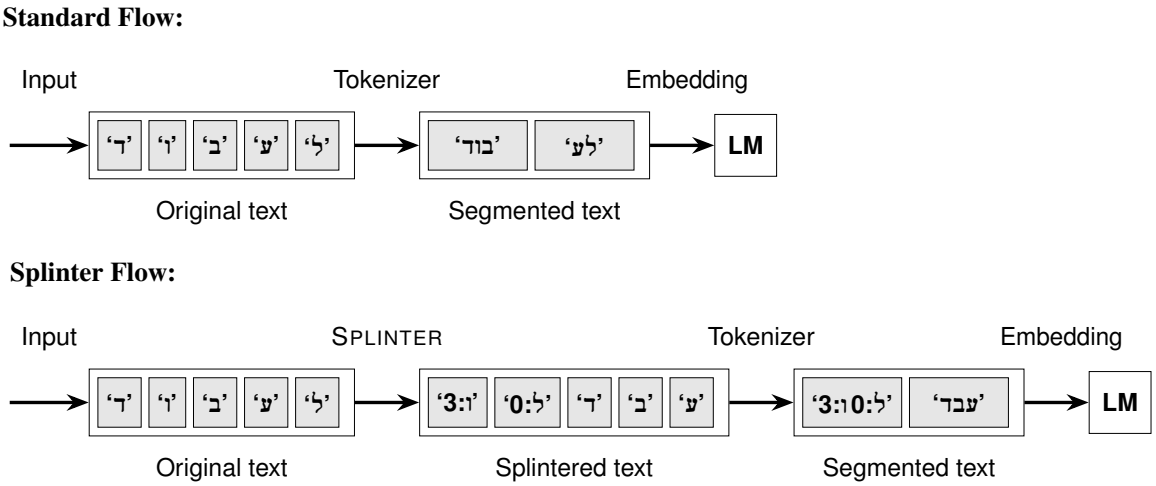

Figure 1: Overview of a Hebrew language model pipeline: standard flow vs. incorporating SPLINTER. Gray boxes are ordered from right to left.

addition, we aimed to develop a tool that would integrate smoothly with existing tokenizers, ensuring seamless adoption without requiring modifications in their implementation. Additionally, the method should be applied only to the intended languages, without affecting the entire text. We also considered that the method should be adaptable to two distinct use cases: (1) models trained primarily in an NCL language, where the majority of the vocabulary belongs to that language (e.g., `DictaBERT` (Shmidman et al., 2023) has a large vocabulary size of 128K); and (2) large-scale multilingual LLMs, where most tokens are allocated to English and only a small portion is left for the NCL language (e.g., GPT-4o, which has approximately 2.3K tokens allocated for Hebrew). Thus, the method should be effective across different vocabulary sizes.

We present SPLINTER, a pre-tokenization step designed to address the challenges of subword tokenization in NCLs. The core idea of the algorithm is that in certain NCLs, many words are formed by embedding root letters into specific morphological templates. For instance, the Hebrew word "**לעבוד**" is derived from the root "**עבד**" placed in the template "ל_ו_". Since there are far fewer templates than roots, template letters tend to appear in specific positions within a word more consistently than root letters do (e.g., for "**לעבוד**": "0:ל","3:ו"). Empirically, we observed that in Hebrew and Arabic, when a word is longer than 3 characters, there is always at least one deletion that, when applied, transitions the word to a different, existing template **while preserving the same root**.[2] For instance, the word "**לעבוד**" transitions to "**לעבד**" when the template letter 'ו' is removed. By repeating this process iteratively, the word eventually reduces to only its root letters. This method can be seen as a way to isolate the root letters from the template.

In this method, NCL words are transformed into a sequence of single-letter reductions, with the goal of rearranging them to better align with existing subword tokenizers. This is achieved by expanding the language's alphabet to include not only its original letters but also single-letter reductions, where each reduction consists of a letter paired with the index from which it was removed.[3] From another perspective, this method explores the impact of expanding a language's alphabet to enhance word representation. We illustrate the high-level application of SPLINTER into the NLP pipeline in Figure 1. Further low-level details are provided in Appendix A.

2.1 SPLINTER Reduction Map Creation

As mentioned above, the SPLINTER pre-tokenization step processes a word by iteratively applying single-letter *reductions*. To achieve this, we must determine which reduction to perform at each iteration. We propose an algorithm that analyzes a given corpus and generates a mapping,

[2]However, not every possible deletion will necessarily result in a valid template. For example, removing ד from לעבוד would produce the non-word לעבו.

[3]In practice, we found that using *negative indices* for the latter half of a word's characters both aligns well with the suffixing nature of certain morphemes and reduces the resulting alphabet by 15%. Not shown in our examples for simplicity.

where the keys represent word lengths and the values are ordered lists of reductions, ranked from most to least frequent.

The algorithm processes a corpus in the target language, constructing a unigram dictionary that maps each word to its frequency. Next, words are grouped by length, and we examine words of length 4 and above (since most Semitic roots are three letters long). For each word of length k, we identify single-letter reductions that result in an existing word of length $k-1$. Each valid reduction is scored by the frequency of the resulting word in the corpus, giving lower scores to rare words, which are more likely to be typos or unique words with no connection to others in the corpus. This score is summed for each word length, per each reduction, so as we iterate through the words, we build a map that tracks, for each word length, which reductions produced valid words, along with their scores. The resulting map is then sorted so that reductions are ranked from most to least frequent.

Following this step, the map is used as a starting point to a second iteration over the corpus. For each word of length k, we attempt to apply reductions from the map, starting with the most frequent one. The first successful reduction that produces an existing word is recorded in a new frequency counter, using the same scoring method as in the previous run, while all other possible reductions for that word are ignored. This step ensures that the most frequent reduction is prioritized, aligning with our goal of removing template letters first, as they tend to appear more frequently than root letters. Pseudocode for the full map creation algorithm is available in Appendix B.

2.2 Tokenizing with Splinters

With the list of reductions for each word length now sorted from highest to lowest scores, we apply it during inference for each word we encounter in a corpus or task data, using a simple search heuristic to produce high-quality relinearizations. We build a scored selection tree where the root is the full character sequence of the word scored at 1.0, and at every node select the b highest-scoring applicable reductions according to the list, keeping the product of reduction scores so far as scores in the next level of nodes. Once either the word length reaches the minimum of 3, or the depth of the tree reaches d, the reduction that started the highest-scoring path is selected and applied, and the process restarts with the new reduced word.[4]

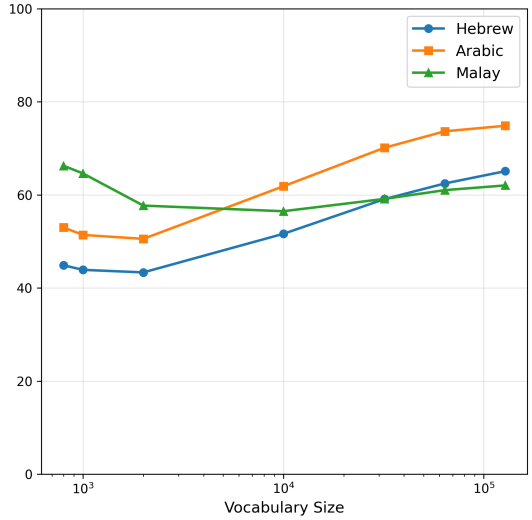

Figure 2: Intersection percent between Vanilla BPE and BPE + SPLINTER by vocabulary size for Hebrew, Arabic, and Malay. Vocabulary size is presented on a logarithmic scale.

As described above, each reduction is encoded as a new *composite character*. This transforms the word into a new representation as a sequence of this enriched alphabet, consisting of the original characters with the addition of the composite ones, which is then used as input for standard subword tokenization methods. From the perspective of the tokenizer and the entire language model, both training and inference operate on this relinearized sequence represented by this new alphabet. In generation mode, a character sequence over the new alphabet is decoded back into reductions, which are then applied sequentially to construct a word.

3 Intrinsic Evaluation

To evaluate the effects of using SPLINTER as a pre-processing step before tokenization, we trained multiple tokenizers on raw text from nonconcatenative languages and their SPLINTER-treated counterparts. We examined the performance of both the bottom-up **BPE** tokenization algorithm, which works on iteratively merging tokens based on corpus co-occurrence statistics, and the top-down **UnigramLM** approach, which starts with a very large vocabulary and iteratively removes from it tokens which contribute minimally to the corpus's likelihood. We train over a wide range of vocabulary sizes in order to assess the utility of SPLIN-

[4] We set $b = d = 3$.

Tokenizer	Type	Cognitive plausibility	Rényi efficiency	Tokens per word	4+ token words	1-char tokens	Distinct Neighbors
BPE	Vanilla	0.157	0.524	**1.146**	**0.53%**	**6.00%**	2674
	SPLINTER	**0.179**	**0.527**	1.165	0.98%	6.81%	**2640**
UnigramLM	Vanilla	0.151	**0.505**	**1.162**	**0.56%**	**9.42%**	2440
	SPLINTER	**0.171**	0.485	1.176	1.00%	12.46%	**2308**

Table 2: Intrinsic benchmark results for Hebrew on a vocab size of 128K. The tokenizers were evaluated using the HeDC4 corpus. **Bold** values indicate better performance between Vanilla and SPLINTER.

TER in multiple scenarios: from multilingual models which can allocate roughly 1,000 tokens for a given language to dedicated monolingual models with room for two orders of magnitude more tokens. We selected three languages for our experiments: Hebrew and Arabic are Semitic languages featuring root-template morphology as discussed above, while Malay is an Austronesian language rich in *circumfixes* and *infixes*—morphemes which break either the stem or the affix when forming the composite inflection. We computed the SPLINTER operations for each language using the reductions map generated from the November 2023 Wikipedia dump of the respective language. The Wikipedia dump sizes were 1.9GB for Hebrew, 3.0GB for Arabic, and 0.4GB for Malay, and were downloaded using the Hugging Face "datasets" library. We trained the tokenizers on the same Wikipedia dump used for SPLINTER training, using Google's SentencePiece library with default settings, except for the tokenizer type (UnigramLM or BPE) and vocabulary size (800, 1K, 2K, 10K, 32K, 64K, 128K).

For direct evaluation of the tokenizer vocabularies, independent of further language model architecture and training, we follow the analytical procedures collected in Uzan et al. (2024), adding pairwise comparative measures from other sources as well.

Vocabulary overlap First, we validate that SPLINTER provides models with vocabularies that are different enough from raw-text tokenizers. Hypothetically, if many common words are learned in full as single tokens from raw text, there is no need for a special pre-processing step to account for an edge case. However, in Figure 2 we show that this is not the case. In all three languages, the maximal vocabulary size of 128K stays at an intersection level below 75%, with the slope of added shared token rate declining to near constant. Moreover, even if we assume a linear extrapolation rate, the intersection rate would only exceed 85% at a vocabulary size of around 780K, which is exceptionally large and is not used even in SOTA English-dominated LLMs like GPT-4o. We note that we used a generous calculation for the intersection rate, as not all tokens in SPLINTER-enhanced tokenizers can be directly compared to those in a regular tokenizer. For instance, a token representing the reduction "0:ל 3:ו" cannot be linearly converted into a standard token. To make the comparison as permissive as possible, we applied the reductions within the token (e.g., converting "0:ל 3:ו" into 'לו') and counted it towards the intersection if the resulting token existed in the raw-text tokenizer's vocabulary. As a result, the actual intersection percentage may be significantly lower than reported. We conclude that SPLINTER-enhanced tokenizers produce significantly different vocabularies at all stages of the vocabulary creation process.

Cognitive plausibility We use the metric introduced in Beinborn and Pinter (2023) to measure the correlation of the tokenizer output with the response time and accuracy of human performance on a lexical decision task. This metric is based on the hypothesis that an effective tokenizer encounters difficulty with character sequences that are also challenging for humans, and vice versa. We use the Hebrew cognitive plausibility dataset (HeLP; Stein et al., 2024) to evaluate both the BPE and UnigramLM tokenizers. Each tokenizer was compared across seven vocabulary sizes, with the Vanilla tokenizer evaluated against the SPLINTER-enhanced tokenizer. Following Uzan et al. (2024), we report the average of the absolute value correlation scores across the four linguistic setups (word/nonword $\times$ accuracy/response time). Higher scores mean better correlation with human performance.

As shown in Table 2 and Table 3, SPLINTER-enhanced tokenizers consistently correlate better with human lexical processing patterns, across all vocabulary sizes in both BPE and UnigramLM. These results suggest that downstream language

Vocab size	Type	Cognitive plausibility	Rényi efficiency	Tokens per word	4+ token words	1-char tokens	Distinct Neighbors
128K	Vanilla	0.157	0.524	**1.146**	**0.53%**	**6.00%**	2674
	SPLINTER	**0.179**	**0.527**	1.165	0.98%	6.81%	**2640**
64K	Vanilla	0.181	0.565	**1.224**	**0.75%**	**7.05%**	4272
	SPLINTER	**0.206**	**0.567**	1.248	1.43%	8.35%	**4188**
32K	Vanilla	0.201	0.610	**1.336**	**1.12%**	**8.76%**	5754
	SPLINTER	**0.223**	**0.612**	1.365	2.04%	10.66%	**5631**
10K	Vanilla	0.196	**0.690**	**1.606**	**2.28%**	**13.26%**	5652
	SPLINTER	**0.226**	0.687	1.651	3.86%	16.56%	**5555**
2K	Vanilla	0.149	**0.760**	**2.137**	**7.44%**	**25.90%**	1855
	SPLINTER	**0.207**	0.756	2.270	11.57%	33.32%	**1815**
1K	Vanilla	0.109	**0.774**	**2.436**	**13.33%**	**34.47%**	925
	SPLINTER	**0.184**	0.763	2.713	22.82%	48.59%	**895**
800	Vanilla	0.102	**0.779**	**2.543**	**16.03%**	**37.70%**	734
	SPLINTER	**0.182**	0.762	2.890	28.70%	54.37%	**705**

Table 3: Intrinsic benchmark results for Hebrew using BPE tokenizer with different vocabulary sizes. The tokenizers were evaluated using the HeDC4 corpus. **Bold** values indicate better performance between Vanilla and SPLINTER.

models trained on SPLINTER output would reach higher scores in morphological segmentation tasks, which we evaluate in §4. We note that, unlike the following metrics, cognitive plausibility does not focus on the tokenizer's effectiveness as a text compression tool, offering a different perspective. Additional UnigramLM results provided in Appendix C.

Token distribution statistics We collected distributional data for the various tokenizers using the following corpora: the HeDC4 corpus (Shalumov and Haskey, 2023) was used in Hebrew experiments looking into vocabulary size and tokenizer type (BPE vs UnigramLM), with a 10% shuffled sample (seed = 42) taken from its original 45GB corpus. For Hebrew's cross-linguistic comparison with Arabic and Malay, we used the respective November 2023 Wikipedia dump of each of the languages. Based on these corpora, we report the Rényi efficiency score (Zouhar et al., 2023), as well as several other surface statistics. We report the Hebrew-specific results in Table 2 and Table 3, and crosslinguistic results in Table 4. Rényi efficiency has been proposed as an indicator of downstream task performance, such as BLEU scores in machine translation. This metric penalizes token distributions that are overly skewed toward either very high- and/or very low-frequency tokens. However, a recent study (Cognetta et al., 2024) suggests that this metric can be manipulated to produce higher scores while degrading actual performance. This highlights the importance of using multiple indicators from different perspectives to make an informed assessment of a tokenizer's potential impact on downstream tasks.

Rényi efficiency scores for Hebrew were consistent across both the HeDC4 and Wikipedia corpora, and in general show minimal differences between SPLINTER tokenizers and Vanilla tokenizers across most tokenization settings. In BPE, the Vanilla tokenizers achieved slightly better results at lower vocabulary sizes ($\leq$2K), while in UnigramLM, the trend was reversed, with SPLINTER tokenizers achieving slightly higher scores at lower vocabulary sizes ($\leq$2K), and the Vanilla tokenizers achieving slightly higher scores at larger vocabulary sizes ($\geq$10K). In Arabic and Malay, SPLINTER's results were again very close to the Vanilla, with SPLINTER tokenizer scores slightly higher in both large and small vocabulary sizes. The overall Rényi efficiency results suggest minimal impact on token distribution, with SPLINTER sometimes slightly improving it and other times slightly reducing efficiency.

For further intrinsic evaluation, we examined three corpus-level indicators of tokenizer compression efficiency: the average number of tokens per word (also known as subword fertility), the percentage of words tokenized into four or more tokens, and the percentage of single-character tokens. All three serve as indicators of compression efficiency, with lower values generally indicating better compression.

Across all corpora, tokenizer types, vocabulary sizes, and languages, the results consistently show

Language	Vocab size	Type	Rényi efficiency	Tokens per word	4+ token words	1-char tokens	Distinct Neighbors
Hebrew	128K	Vanilla	0.509	**1.119**	**0.57%**	**6.01%**	1463
		SPLINTER	**0.511**	1.134	0.92%	6.61%	**1460**
	2K	Vanilla	**0.779**	**2.149**	**9.22%**	**26.34%**	1853
		SPLINTER	0.777	2.306	13.65%	33.81%	**1805**
Arabic	128K	Vanilla	0.427	**1.134**	**0.55%**	**7.54%**	**1444**
		SPLINTER	**0.430**	1.158	1.07%	7.84%	1520
	2K	Vanilla	0.736	**2.117**	**11.25%**	**29.37%**	1824
		SPLINTER	**0.744**	2.276	16.70%	37.91%	**1784**
Malay	128K	Vanilla	0.471	**1.088**	**0.55%**	**4.45%**	**337**
		SPLINTER	**0.479**	1.135	1.56%	5.98%	354
	2K	Vanilla	0.756	**2.150**	**14.65%**	**29.23%**	1215
		SPLINTER	**0.770**	2.724	28.79%	43.37%	**1055**

Table 4: Intrinsic benchmark results using BPE tokenizer for Hebrew, Arabic and Malay on a vocab sizes of 2K and 128K. The tokenizers were evaluated using the Wikipedia corpus of their respective language. **Bold** values indicate better performance between Vanilla and SPLINTER.

that adding SPLINTER to the tokenizer reduces its compression efficiency. This result is important, as in generative LMs, for instance, less efficient compression requires more iterations to generate the same text, leading to higher computational costs. Therefore, this trade-off should be considered when applying SPLINTER in an LLM tokenizer. That being said, Schmidt et al. (2024) found that tokenization should not be viewed principally from the compression perspective. Improved compression does not always correlate with better downstream task performance, and in some cases, it may even degrade it. This again emphasizes the importance of considering multiple perspectives when assessing tokenization quality.

Contextual coherence The next aspect we examine is the contextual coherence (Yehezkel and Pinter, 2023) of the tokens produced by each tokenizer, as measured by the number of distinct neighbors each token encounters within a window of k tokens from each side (we choose $k = 2$). This measurement is motivated by the main downstream application scenario of vocabularies—contextualized embeddings in language models. The fewer contexts a token appears in, the better a model is likely to learn a meaningful embedding for it over a corpus, as it offers better differentiation between token environments. Figure 3 displays the number of neighbors for the top 200 tokens in each tokenizer as ranked according to this number. We present the average number of distinct neighbors across the vocabulary in Tables 2, 3, and 4 as a measure of efficiency—not for text compression, but for downstream LM training. As shown in these tables, SPLINTER-based tokenizers consistently produce fewer distinct neighbors than Vanilla in Hebrew, regardless of vocabulary size, tokenization algorithm, or corpus. However, when evaluated on Wikipedia corpora, which were also used for training the tokenizers themselves, the differences were less pronounced. SPLINTER achieved a lower average number of distinct neighbors only at the 2K vocabulary size, while at 128K, its results were nearly identical to the baseline in Hebrew, and in Arabic and Malay, the baseline outperformed SPLINTER. These mixed results suggest that the impact of SPLINTER on downstream tasks may depend on vocabulary size, with its advantages being more evident in smaller vocabularies dedicated to an NCL language, while in larger vocabulary sizes, the baseline tokenizer may perform better.

4 Language Modeling with Splinters

As noted above, standard subword tokenizers are suboptimal for nonconcatenative languages such as Hebrew and Arabic. Thus, we proceed now to evaluate SPLINTER's impact on downstream NLP tasks for Hebrew. Specifically, we evaluate Prefix Segmentation and Syntactic Parsing (Sade et al., 2018; Bareket and Tsarfaty, 2021; Zeldes et al., 2022), and Question Answering (Cohen et al., 2023).

We begin with the open `DictaBERT` BERT-base model (Shmidman et al., 2023), which delivers current SOTA-level performance on the aforementioned tasks (Shmidman et al., 2024) in Hebrew. We then pre-train a new BERT-base model using the same corpus and training parameters as `DictaBERT`, with the only modification being the

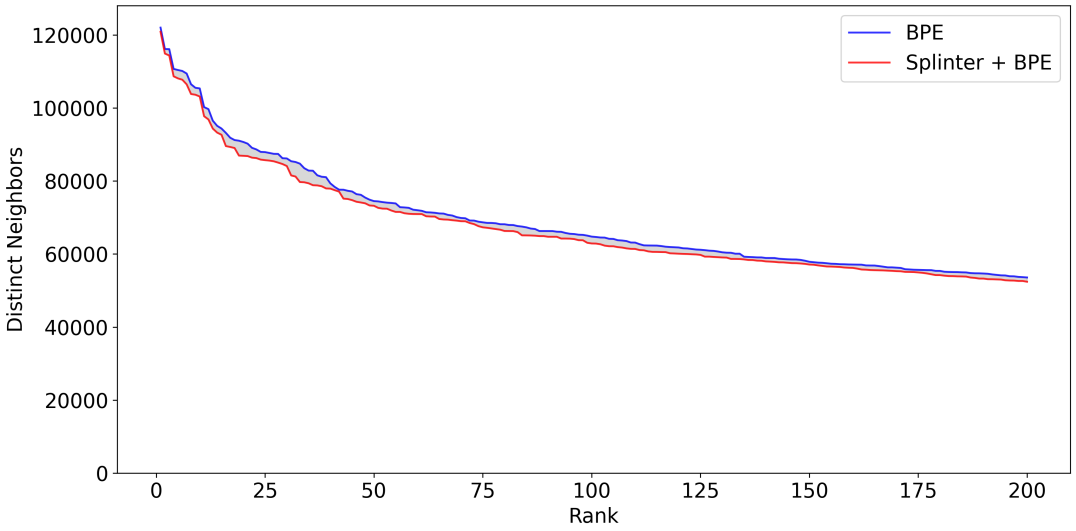

Figure 3: Distinct neighbor counts for top 200 tokens in BPE and SPLINTER + BPE for a window of 2 on each side, vocabulary size 128K, HeDC4 corpus.

Model	QA (F1)	QA (EM)	Syntax (LAS)	Seg (Acc)
DictaBERT	72.9	63.6	**89.0**	99.1
SPLINTER	**74.4**	**65.4**	**89.0**	**99.3**

Table 5: Performance comparison of the existing SOTA Hebrew BERT (DictaBERT) with our newly-pretrained SPLINTER-based Hebrew BERT, across three downstream Hebrew NLP tasks.

SPLINTER-processed tokenization. We then fine-tune the new base model for the aforementioned tasks, and evaluate performance vis-a-vis fine-tuning the original `DictaBERT`. We follow the same task parameters defined by Shmidman et al. (2024) and Shmidman et al. (2023) for the three tasks. We present the results in Table 5.

Results Regarding both of the Question-Answer benchmarks, we find that the SPLINTER-based model provides a substantial boost in performance. We believe that this indicates that the SPLINTER-based tokenization provides the model with a substantially stronger ability to process and understand the message of a Hebrew text, and thus to achieve superior performance on this high-level textual challenge. On the sentence-level task of syntactic parsing, the performance of the SPLINTER-based model is essentially the same as the existing `DictaBERT` model, indicating diminished advantage for manipulation at the character level. However, for the nearly-saturated task of labeling prefix segmentation at the character level, SPLINTER-based tokenization provides over 20% reduction in errors, highlighting the effectiveness of the data-driven pattern-finding algorithm it employs.

Example An illustrative example in which SPLINTER's tokenization architecture allows it to succeed where `DictaBERT` fails is the following question from the aforementioned QA corpus, regarding the date of a certain archaeological excavation. The relevant part of the input text is comprised of the following three sentences (cited here in English translation):

"In 1913 he purchased the tract of land that covers most of the eastern slope of the City of David in Jerusalem, and persuaded the French Jewish archaeologist Raymond Weill to conduct excavations there. This was done in response to the scandalous excavation of Montagu Parker in the City of David in 1911. Rothschild returned and financed another season of excavations, in 1923–1924, under Weill's direction."

The following question is then posed to the system: "In what year were the first excavations conducted in the City of David?"

`DictaBERT` incorrectly answers 1923 (as per the third sentence), while SPLINTER correctly answers 1911 (as per the second sentence). `DictaBERT`'s failure to pull the correct answer from the second sentence likely stems from the fact that in the origi-

nal Hebrew of that sentence, the words "excavation of Montague Parker" are phrased using the singular form "excavation", with a suffixed possessive pronoun (חפירתו *xafirato*). This word differs from the non-suffixed plural term "excavations" used in the question (חפירות *xafirot*). Crucially, the difference between the terms is not just the suffixed ו *o* at the end, but also a letter from the middle of the base term. It is precisely discrepancies such as these that SPLINTER aims to solve. It stands to reason that `DictaBERT`'s incorrect answer stems from its inability to see the connection between these two terms; thus, it was unable to understand the relevance of the second sentence, and instead took its incorrect answer from the subsequent sentence, which includes an exact match for the term "excavations". In contrast, thanks to its new tokenization architecture, SPLINTER recognizes the connection between the two disparate terms and correctly answers "1911".

5 Conclusion

In this work, we introduced SPLINTER, a novel pre-processing method for subword tokenizers designed to improve downstream performance on nonconcatenative languages (NCLs). By applying an iterative reduction process, SPLINTER restructures words in a way that better aligns with existing subword tokenizers. Our approach was designed with key constraints in mind: ensuring lossless transformation, compatibility with existing tokenization frameworks, and applicability across different vocabulary sizes and model types, whether for an NCL, a single-language LM, or a multilingual model based on English with a limited number of tokens allocated for NCL languages.

Through intrinsic evaluations, we demonstrated that SPLINTER-enhanced tokenizers exhibit distinct vocabulary distributions compared to Vanilla tokenizers. Cognitive plausibility metrics indicated that SPLINTER improves alignment with human-like lexical processing, while our analysis of compression-related metrics revealed that SPLINTER trades off slight reductions in compression efficiency for potentially better linguistic representation.

Our downstream evaluation highlights SPLINTER's impact, particularly on higher-level NLP tasks such as question answering and on character-critical tasks such as prefix segmentation. The intermediate syntactic level appears to be less affected by the nonconcatenativity of Hebrew text.

In future work, we will extend the downstream evaluation to Arabic and other Semitic languages, as well as more languages exhibiting non-templatic nonconcatenative phenomena. Additionally, we plan to evaluate the performance of a large multilingual generative model on various tasks after incorporating SPLINTER, examining its effectiveness in a broader linguistic context.

Limitations

Rearranging text in order to improve representation of nonconcatenative features is a hard high-level problem, and we believe our work is a first step towards remedying this inherent mismatch between modeling and language data. However, our concrete algorithm is still not universally-applicable, as shown by the difference between results on Semitic languages and on Malay. Primarily, we attribute this to the property where each single-character pruning action must result in a valid corpus word, mostly limiting the scope of linearization to templatic morphology rather than also including infixation and circumfixation.

In addition, the increase in performance comes at the cost of less efficient token sequences, as found in our fertility analysis. Overcoming this tradeoff is important for lowering the costs of running LLMs on low-resource languages, already lagging behind their high-resource counterparts.

References

Dan Bareket and Reut Tsarfaty. 2021. Neural modeling for named entities and morphology (NEMO2). *Transactions of the Association for Computational Linguistics*, 9:909–928.

Lisa Beinborn and Yuval Pinter. 2023. Analyzing cognitive plausibility of subword tokenization. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 4478–4486, Singapore. Association for Computational Linguistics.

Marco Cognetta, Vilém Zouhar, Sangwhan Moon, and Naoaki Okazaki. 2024. Two counterexamples to tokenization and the noiseless channel. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 16897–16906, Torino, Italia. ELRA and ICCL.

Amir Cohen, Hilla Merhav-Fine, Yoav Goldberg, and Reut Tsarfaty. 2023. HeQ: a large and diverse Hebrew reading comprehension benchmark. In *Find-*

ings of the Association for Computational Linguistics: EMNLP 2023, pages 13693–13705, Singapore. Association for Computational Linguistics.

Omri Keren, Tal Avinari, Reut Tsarfaty, and Omer Levy. 2022. Breaking character: Are subwords good enough for mrls after all? *arXiv preprint arXiv:2204.04748*.

Bilal Khaliq and John Carroll. 2013. Induction of root and pattern lexicon for unsupervised morphological analysis of Arabic. In *Proceedings of the Sixth International Joint Conference on Natural Language Processing*, pages 1012–1016, Nagoya, Japan. Asian Federation of Natural Language Processing.

Stav Klein and Reut Tsarfaty. 2020. Getting the ##life out of living: How adequate are word-pieces for modelling complex morphology? In *Proceedings of the 17th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 204–209, Online. Association for Computational Linguistics.

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.

Danit Yshaayahu Levi and Reut Tsarfaty. 2024. A truly joint neural architecture for segmentation and parsing. *arXiv preprint arXiv:2402.02564*.

Shoval Sade, Amit Seker, and Reut Tsarfaty. 2018. The Hebrew Universal Dependency treebank: Past present and future. In *Proceedings of the Second Workshop on Universal Dependencies (UDW 2018)*, pages 133–143, Brussels, Belgium. Association for Computational Linguistics.

Craig W Schmidt, Varshini Reddy, Haoran Zhang, Alec Alameddine, Omri Uzan, Yuval Pinter, and Chris Tanner. 2024. Tokenization is more than compression. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 678–702, Miami, Florida, USA. 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.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

Vitaly Shalumov and Harel Haskey. 2023. Hero: Roberta and longformer hebrew language models. *arXiv:2304.11077*.

Shaltiel Shmidman, Avi Shmidman, and Moshe Koppel. 2023. Dictabert: A state-of-the-art bert suite for modern hebrew.

Shaltiel Shmidman, Avi Shmidman, Moshe Koppel, and Reut Tsarfaty. 2024. MRL parsing without tears: The case of Hebrew. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 4537–4550, Bangkok, Thailand. Association for Computational Linguistics.

Roni Stein, Ram Frost, and Noam Siegelman. 2024. Help: The hebrew lexicon project. *Behavior Research Methods*, 56(8):8761–8783.

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.

Shaked Yehezkel and Yuval Pinter. 2023. Incorporating context into subword vocabularies. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pages 623–635, Dubrovnik, Croatia. Association for Computational Linguistics.

Amir Zeldes, Nick Howell, Noam Ordan, and Yifat Ben Moshe. 2022. A second wave of UD Hebrew treebanking and cross-domain parsing. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 4331–4344, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Vilém Zouhar, Clara Meister, Juan Gastaldi, Li Du, Mrinmaya Sachan, and Ryan Cotterell. 2023. Tokenization and the noiseless channel. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5184–5207, Toronto, Canada. Association for Computational Linguistics.

A Implementation Details

A.1 Alphabet Encoding

To support the expanded set of characters introduced by SPLINTER, which includes the original alphabet along with symbols representing single-letter reductions, we needed an alphabet capable of handling a large number of unique symbols. In Hebrew, this expanded alphabet comprised 252 characters, while for Arabic and Malay, it grew to 400 and 460 characters, respectively. Since the Unicode character sets for these languages do not offer enough distinct symbols, we mapped each new character to a unique Chinese letter, leveraging the large character set available in the Chinese writing system. This was done as a workaround, as attempts to use the Unicode Private Use Areas (PUA) with the SentencePiece library were unsuccessful. This approach allowed the tokenization process to remain seamless from the language model's perspective, as it processed the input as Chinese text, effectively encoding the original text.

A.2 Converting the Corpus to Unigram Frequencies

The corpus undergoes several preprocessing steps before generating the unigram frequency counts. First, all diacritics are removed from the text (in Hebrew only). Next, the text is split into words using the following regex pattern:

```
\.|\s|\n|-|,|:|"|\(|\)
```

Words that appear fewer than 10 times in the corpus are then discarded, along with any words containing letters from other languages. Additionally, we normalized final and non-final Hebrew letters to maintain consistency in root-based word connections. Specifically, all final letters were replaced with their non-final forms, and vice versa when applicable (e.g., in a case of a non-final form in a final position of the word. Mostly in borrowed words like "קטשופ" "ketchup"). This adjustment helps preserve morphological relationships, such as between "הולך" ("he is walking") and "הולכים" ("they are walking"), both derived from the root "הלכ". After this transformation, these words become "הולכ" and "הולכימ", both clearly retaining the root "הלכ". The reverse transformation ensures that distinctions between different word groups are still maintained. It's worth noting that Arabic also has final and non-final letter forms, but their selection occurs automatically based on context. From a Unicode perspective, both forms share the same underlying character, eliminating the need for manual conversion.

B SPLINTER Algorithm

Pseudocode for the SPLINTER map creation algorithm is presented in Algorithm 1.

C SPLINTER UnigramLM results

Intrinsic benchmark results for Hebrew using UnigramLM tokenizer on HeDC4 corpus are presented in Table 6.

Algorithm 1 High-level algorithm for training SPLINTER.

```
function TRAINSPLINTER(corpus)
    freqMap ← GetWordFrequenciesFromCorpus(corpus)
    reductions ← InitializeEmptyReductionsMap()
    for length ← 4 to maxWordLength do
        for word in freqMap[length] do
            for position in word do
                permutation ← GetWordWithoutLetter(word,position)
                if permutation ∈ freqMap[length - 1].keys then
                    reduction ← Reduction(position,word[position])
                    frequency ← freqMap[length - 1][permutation]
                    reductions[length][reduction] += frequency
                end if
            end for
        end for
    end for
    sortedReductions ← sortReductionsByScoreDesc(reductions)
    selectedReductions ← InitializeEmptyReductionsMap()
    for length ← 4 to maxWordLength do
        for word in freqMap[length] do
            for reduction in sortedReductions[length] do
                Extract (position, letter) from reduction
                if word[position] == letter then
                    permutation ← GetWordWithoutLetter(word,position)
                    if permutation ∈ freqMap[length−1].keys then
                        frequency ← freqMap[length − 1][permutation]
                        selectedReductions[length][reduction] += frequency
                        Break
                    end if
                end if
            end for
        end for
    end for
    Return selectedReductions
end function
```

Vocab size	Type	Cognitive plausibility	Rényi efficiency	Tokens per word	4+ token words	1-char tokens	Distinct Neighbors
128K	Vanilla	0.151	**0.505**	**1.162**	**1.00%**	**9.42%**	2440
	SPLINTER	**0.171**	0.485	1.176	0.56%	12.46%	**2308**
64K	Vanilla	0.180	**0.522**	**1.243**	**0.88%**	**11.50%**	3907
	SPLINTER	**0.194**	0.495	1.261	1.65%	16.21%	**3640**
32K	Vanilla	0.191	**0.526**	**1.363**	**1.46%**	**14.41%**	5322
	SPLINTER	**0.208**	0.496	1.391	2.74%	21.48%	**4931**
10K	Vanilla	0.177	**0.536**	**1.663**	**3.62%**	**21.26%**	5267
	SPLINTER	**0.213**	0.517	1.713	6.53%	31.63%	**5064**
2K	Vanilla	0.136	0.590	**2.250**	**11.65%**	**33.40%**	1824
	SPLINTER	**0.196**	**0.618**	2.424	20.70%	51.04%	**1776**
1K	Vanilla	0.127	0.618	**2.604**	**21.22%**	**43.92%**	917
	SPLINTER	**0.185**	**0.659**	2.877	32.48%	63.21%	**881**
800	Vanilla	0.126	0.629	**2.730**	**25.33%**	**47.98%**	726
	SPLINTER	**0.177**	**0.673**	3.060	37.71%	68.34%	**693**

Table 6: Intrinsic benchmark results for Hebrew using UnigramLM tokenizer with different vocabulary sizes. The tokenizers were evaluated using the HeDC4 corpus. **Bold** values indicate better performance between Vanilla and SPLINTER.