ReBPE: Iteratively Improving the Internal Structure of a Structured Tokeniser by Mining its Internal Structure

Thomas Bauwens and **Miryam de Lhoneux**
LAGOM·NLP, Department of Computer Science, KU Leuven
firstname.lastname@kuleuven.be

Abstract

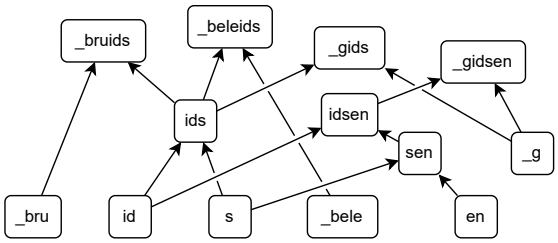
Recent work has explored pruning merges from BPE subword tokenisers using corpus data as a signal for which merges to prune. We argue that because a BPE tokeniser contains a rich data structure on top of its vocabulary set, this in itself can be used as a guide to modify its merges such that segmentations become more desirable. We apply this argument to one of those pruning algorithms, BPE-knockout (Bauwens and Delobelle, 2024), by introducing a new *reification* step that suggests new merges by inspecting the effects left by pruning. By alternating both processes iteratively until convergence, we get a new BPE tokeniser, **ReBPE**, which outperforms the original BPE-knockout algorithm on morphological alignment in all 14 languages tested by over 11% F_1 on average.

1 Introduction

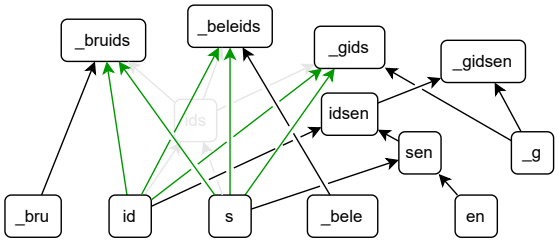
Various algorithms exist to train subword tokenisers, which are mappings from the infinite space of input texts encountered by a language model to sequences of units (*tokens*) taken from a finite set (*vocabulary*) of possible units (*types*).

Some of those algorithms need little more than the vocabulary itself to segment text; for example, a ULM tokeniser (Kudo, 2018) consists of a flat set of vocabulary types with associated unigram probabilities. Others have more structure, and contain rich information about how types relate to each other: in a BPE tokeniser (Sennrich et al., 2016), every type that isn't a character is linked to two other types that were concatenated to construct it. In a Morfessor tokeniser (Creutz and Lagus, 2007), the same relationships hold, except they are constructed top-down, splitting rather than merging.

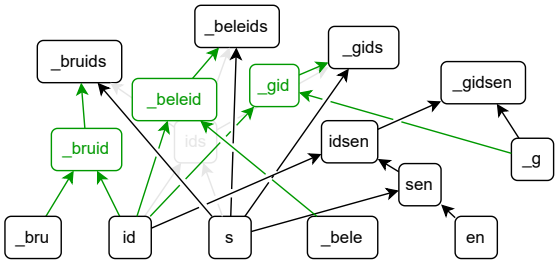
It's not obvious that all such structural information is recruited by the tokeniser, despite its existence. For example, even though BPE segments text by building trees of tokens, the knowledge learnt during vocabularisation forms a graph, not a tree



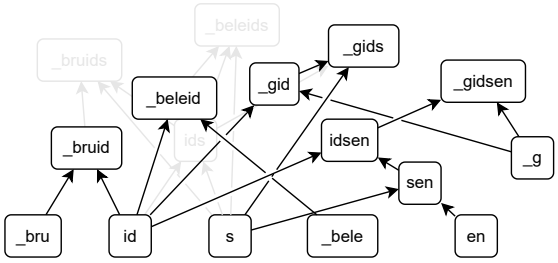
(a) BPE



(b) + knockout (⇒ BPE-knockout)



(c) + knockout + reify (⇒ ReBPE limited to 1 iteration)



(d) + knockout + reify + knockout

Figure 1 – Application of 1.5 iterations of ReBPE to an example BPE merge graph in Dutch. If knockout would be run directly after case (b), the types [*_bru, id, s*] and [*_bele, id, s*] would just become fully unrelated.

(Bauwens and Delobelle, 2024): if a type is a parent to multiple other types – e.g. two compounds formed from the same stem – then those types are "half-siblings" of each other, which is not exploited

Findings of the Association for Computational Linguistics: EACL 2026, pages 4075–4090
March 24-29, 2026

during segmentation. This suggests that rather than using external data, refining a tokeniser's segmentations may be possible just by getting inspiration from its own, current structure.

A handful of works (compared in §2) have explored modifying the structure of the BPE tokeniser using corpus data. They all remove types. All but one of them target the relatively few "scaffold" types in the vocabulary, i.e. meaningless types formed by the BPE vocabulariser merely as stepping stones to build up larger types, rarely occurring themselves. They mark these by thresholding types' corpus frequency in some way, and break them down at some point in BPE's merge process.

BPE-knockout (Bauwens and Delobelle, 2024) is the one exception which marks a much[1] broader range of types by judging the merges that create them instead. Pair merges (x, y) that too often violate token boundary constraints demonstrated implicitly by a corpus of desirable segmentations get permanently removed from the tokeniser. To keep children (xy, z) of removed types accessible, their merges integrate the removed merge and become "tuple merges" (x, y, z). In effect, knocked-out types can no longer appear *unless immediately* merged into an even bigger type.

As pointed out by the authors in their limitations section, the algorithm may not quite achieve what it purports to. They raise the example of a Dutch modifier *bruids*, consisting of *bruid* ("bride") and a possessive *s*, used in compounds such as *bruidsjurk* ("wedding dress", lit. "dress [*jurk*] of [*s*] a bride [*bruid*]"). With morphological alignment as the desirability constraint, BPE-knockout discovers that the *d* should not stick to the *s*, for which apparently the merge *id+s→ids* is to blame. *ids* is knocked out (see Figure 1b), and its merge is absorbed into merges that formed its children, like *g+id+s→gids* ("guide"). Yet, for the type *bruids*, this is ineffectual: the *d* still sticks to the *s*, except now it is the triplet merge *bru+id+s* that is to blame (originally *bru+ids*) rather than the pair merge *id+s*. Running knockout a second time to detect and eliminate the triplet *bru+id+s* is also unproductive, because this loses the useful link between *bru* and *id*.

Given the token sequence [*bru, id, s*], the most desirable scenario would be to have a pair merge *bru+id* apply and nothing else, to end up with [*bruid, s*]. We can achieve the "nothing else" using knockout: if we first apply *bru+id* to the triplet merge *bru+id+s* (Figure 1c) and do the same in any input sequence, it becomes a merge *bruid+s* which, when knocked out (Figure 1d), leaves the token *bruid* behind exactly as desired.

In other words, *tuple merges in a BPE-knockout tokeniser suggest merges that could improve the merges in that same tokeniser*, rather than needing corpus data to suggest those new merges.

Based on the above, in this paper, we:

- define a new procedure, *reification*, that mines tuple merges for new binary merges to add to a BPE-knockout tokeniser;
- apply this iteratively in a new vocabularisation algorithm, *reifying BPE* (**ReBPE**), which alternates between knockout to remove merges and reification to contrive new merges to try;
- show that ReBPE adheres more to morphology than applying only BPE-knockout.

2 Related Work

2.1 Removing types from a BPE tokeniser

Four different algorithms have been published to remove a set of types marked as undesirable from a BPE tokeniser's output. Two of them modify the structure of the BPE tokeniser to accomplish this.

BPE-knockout Bauwens and Delobelle (2024) proposed bypassing undesirable types in the BPE merge graph, as discussed in §1. Such types can then no longer appear as a token in the output, and the tokens they were built from either stay unmerged, get picked up separately by merges with the surrounding tokens, or get used simultaneously by a tuple merge. Given a corpus where words are split desirably, a type is marked when its merge is responsible for crossing such a split the majority of the time it is applied. Chizhov et al. and Lian et al. allege that this semi-supervision is restricted to morphological data and synthetic languages respectively, yet *any* segmentation constraint could be absorbed by BPE-knockout, including Chinese word boundaries or another tokeniser's output.[2]

[1] To illustrate: according to the respective appendices of Bauwens and Delobelle (2024) and Chizhov et al. (2024), in a 32ki vocabulary, BPE-knockout$_{0.5}$ eliminates over 4400 types, whilst PickyBPE$_{0.9}$ eliminates under 700, more than 6× fewer.

[2] Chizhov et al. also claim that BPE-knockout's purpose is to remove scaffold tokens, but Bauwens and Delobelle point out exactly as a limitation that it *doesn't* do so, since scaffolding that builds up a morpheme doesn't itself violate a morpheme boundary and is thus its merges aren't detected.

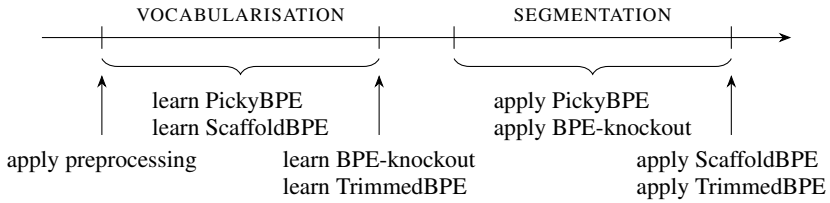


Figure 2 – Lifecycle of a BPE tokeniser, and when various algorithms in the literature operate.

TrimmedBPE Cognetta et al. (2024) also propose to first train a BPE tokeniser as usual, and then apply it to a secondary corpus to mark types to remove. Unlike BPE-knockout, their criterion is instead that a type's absolute frequency doesn't exceed a set threshold. They do not modify the merge list; instead, at segmentation time, they first apply the original BPE tokeniser, and then, for each of the marked tokens among the resulting segmentation, apply a *recursive decomposition* function that walks back down the merge tree to find the largest tokens that are not marked.

ScaffoldBPE Lian et al. (2025) also propose applying recursive decomposition at segmentation time, but they mark types differently. As usual, during each iteration of BPE vocabularisation, the most frequent adjacent pair of types in the training corpus is merged into a new type, until some stopping condition (e.g. a desired vocabulary size) is reached. Let f measure frequencies in the corpus after a merge (x, y) takes place. ScaffoldBPE first finds the next pair (a, b) to be merged, and checks whether the frequency of the parents of the previous merge, x and y, was depleted so much by it that now $f(x) \leq f(a, b)$ and/or $f(y) \leq f(a, b)$. They are marked accordingly.

PickyBPE Chizhov et al. (2024) also propose marking types during vocabularisation, but using $\lambda \cdot f(xy)$, $\lambda \in [0, 1]$, rather than $f(a, b)$, as shown in §A.4. The presence of the hyperparameter is contentious: they criticise Cognetta et al. (2024)'s absolute threshold for being corpus-dependent, but are in turn criticised by Lian et al. (2025) for having to tune a hyperparameter at all (and indeed, no universally superior λ was found). The biggest difference with ScaffoldBPE is that PickyBPE doesn't use marking for recursive decomposition: instead, it generalises BPE's merge list to an "event list" containing merge events $(ab, cd) \to abcd$ and split events $ab \to (a, b)$, recorded right after each merge for either or both of its parent types, if marked.

2.2 Adding types to a BPE tokeniser

To our knowledge, only one paper has attempted adding subword types to a BPE tokeniser that has already undergone vocabularisation, at least without jettisoning the merge list altogether: Delobelle et al. (2022) use a domain-adapted BPE tokeniser as their source of new merges, appending its merges to the end of the merge list of an old BPE tokeniser. Problematically, there is no guarantee that such new merges will ever be used, and Bauwens (2023) proved that applying the combined BPE tokeniser *to its own vocabulary* would sometimes result in sequences of more than one token.

3 ReBPE

We now give the outline for an iterative algorithm to apply to an existing BPE tokeniser to make it align more with a corpus $\mathcal{D}$ of desirable segmentations. Each iteration, it applies BPE-knockout, repairs unforeseen damage done by doing so, and finds binary unexplored merges to add.

3.1 Reparation

In §A.1, we prove that there exist BPE tokeniser structures in which a tuple merge created by BPE-knockout is impossible to apply, even though the purpose of tuple merges was precisely to not lose a knocked-out type's descendants.

To unblock a tuple merge $(x_1, x_2, x_3, \ldots) \to s$, with rank r, we first apply merges 1 to $r - 1$ to the string s to get a new token sequence $[y_1, y_2, \ldots]$. Then, in the merge list, we replace the old merge by $(y_1, y_2, \ldots) \to s$. Algorithm 2 shows this in blue.

3.2 Reification

Once all tuple merges (from having applied BPE-knockout first) are repaired, they are up for consideration during a second round of BPE-knockout blame computation. Now, a problem occurs that didn't in the first round: a tuple merge of n tokens, of the form $(a, b, c, d, \ldots)$, contracts $n - 1$ token boundaries, rather than just 1. That means, in the

worst case, $n-2$ justifiable token contractions can be lost due to 1 with high blame.

To reduce the risk of this happening, we should reduce how many tokens each tuple merge contains. Thus, we should merge as many token pairs as possible inside tuple merges, so that the specific blameworthy pair(s) don't affect the rest. We can turn merges of the form $(a, b, c, d, e, f, \ldots) \to s$ into $(ab, cd, ef, \ldots) \to s$ for this purpose by inserting binary merges right in front of the tuples.

Algorithm 2 shows the discovery of these binary merges in red, and their insertion – *reification* – in green. Whether these binary merges should be able to expand the vocabulary, is up to configuration.

3.3 Iteration

Since neither reparation nor reification are guaranteed to produce desirable merges, knockout should be run on the resulting tokeniser. To guarantee that iterating these three steps eventually makes the BPE graph converge, it suffices to disallow reification from forming merges that were knocked out.

The start and end of the entire iterative process is necessarily a round of knockout, because reification cannot be done without the presence of tuple merges, whilst not doing knockout at the end would leave the latest new merges unvetted.

3.4 Annealing

The algorithms in §2 for pruning scaffold types are all based on the observable drop in frequency once a scaffold type *is* consumed in a merge. What they don't detect are meaningless types that *would* be consumed by such a merge *if* the vocabulary were bigger: e.g., the 8ki PickyBPE$_{0.9}$ tokeniser of Chizhov et al. (2024) still contains the type *_Afric* because the vocabulariser never got to *_Africa*.

To find the merges that would have been formed from scaffolding, we can use the complement of the blame metric used by BPE-knockout: rather than finding existing merges that cross given boundaries as much as possible, we look for non-existing merges that cross given boundaries as little as possible (e.g. *_Afric+a*). These are guaranteed not to need knockout afterwards, and are intended to boost split precision rather than split recall. We call this extra technique for adding merges *annealing*.

Since Bauwens and Delobelle (2024) showed that knockout lengthens token sequences, annealing (at least with binary merges) is best done even before the first round of knockout.

We use the name *ReBPE* to refer to the full BPE post-processing procedure described above (and summarised in Algorithm 1) with annealing, knockout, reparation and reification.

4 Experiments

We now train BPE tokenisers in 14 different European languages, and use ReBPE to let them absorb boundaries prescribed by a reference dataset of morphological segmentations.

Corpora We combine Fineweb (Penedo et al., 2024) and Fineweb-2 (Penedo et al., 2025) to train one BPE tokeniser for each language[3] we have a reference for, across the equivalent of 30 million English examples (details about sizes in §B).

Reference We use the morphological reference segmentations of both inflections and derivations in MorphyNet (Batsuren et al., 2021). We follow the standard practice of considering tokenisation as a binary classification problem of split positions as per Kurimo et al. (2006), and measure precision, recall and F_1 micro-averaged across the unique words in the reference. As argued by Bauwens and Delobelle, since the goal here is precisely to memorise the data used for post-processing, we do not do any holdout (i.e. the references used to inform knockout are also those used for evaluation). We do limit the dataset size (at most 250 000 examples per language, which the majority of MorphyNet supports) because otherwise more memorisation pressure is put on some tokenisers despite the same initial finite memory capacity of $|V| = 32$ki.

Hyperparameters We create one BPE tokeniser with $|V| = 2^{15} = 32$ki on each of the 14 Fineweb corpora. We use the same preprocessor for each of these, and use it for both vocabularisation and segmentation. Details are again in §B.

Software For both BPE vocabularisation and morphological evaluation, we use the *Tokeniser Toolkit (TkTkT)* (Bauwens, 2025) Python package.

5 Results

5.1 Main

Table 1 shows the experimental results. As hypothesised, ReBPE outperforms BPE-knockout across

[3]We interleave the corpora for Serbian, Croatian and Bosnian, all standard varieties of the nameless pluricentric South Slavic language whose varieties are united by their separate standardisations based on the Štokavian dialect (Sussex and Cubberley, 2006), and refer to this corpus as "S/C/B".

	$\lvert V\rvert$			Pr			Re			F_1		
	BPE	BPE-k	ReBPE	BPE	BPE-k	ReBPE	BPE	BPE-k	ReBPE	BPE	BPE-k	ReBPE
Catalan	32 768	29 485	34 653	21.70	42.69	61.49	24.64	67.84	71.11	23.07	52.40	65.95
Czech	32 768	30 844	45 925	8.61	25.12	36.80	14.08	50.62	56.92	10.68	33.58	44.70
English	32 768	30 348	44 508	14.21	28.22	36.12	29.99	71.67	70.92	19.29	40.50	47.86
Finnish	32 768	31 526	48 535	10.94	20.46	28.21	22.29	46.66	48.90	14.67	28.45	35.78
French	32 768	29 956	43 252	19.02	29.66	45.19	36.41	63.49	64.29	24.99	40.43	53.07
German	32 768	30 803	47 570	11.90	18.97	25.68	29.81	51.62	51.71	17.01	27.75	34.31
Hungarian	32 768	29 968	45 761	33.81	44.86	64.57	56.16	84.76	86.26	42.21	58.67	73.85
Italian	32 768	31 641	48 957	8.90	18.87	29.12	16.56	40.03	40.43	11.58	25.65	33.86
Polish	32 768	31 424	38 876	14.77	27.40	35.83	26.15	57.56	59.97	18.88	37.13	44.86
Portuguese	32 768	29 595	42 082	17.97	32.36	53.26	31.15	64.90	66.79	22.79	43.19	59.26
Russian	32 768	31 719	48 099	15.31	28.92	49.40	18.45	43.76	57.95	16.74	34.82	53.34
S/C/B	32 768	31 843	34 265	18.16	39.57	48.12	22.11	63.96	75.00	19.94	48.90	58.63
Spanish	32 768	30 772	47 368	28.95	39.37	57.58	42.92	67.92	68.30	34.58	49.85	62.48
Swedish	32 768	30 118	38 169	21.71	45.69	61.68	26.41	77.00	79.66	23.83	57.35	69.53
Avg. diff.		−2051	+10 662		+14.01	+27.65		+32.47	35.79		+19.88	+31.23

Table 1 – Vocabulary sizes and micro-averaged morphological alignment of 32ki BPE tokenisers and their derived BPE-knockout and ReBPE tokenisers across MorphyNet. The last row shows the average increase relative to BPE.

all 14 languages and metrics (except English Re, -0.75%), by an absolute +7% to +18% in F_1 score and over +11% on average. In half of all languages, precision jumps by at least +15%, with Portuguese at just under +20%. Recall jumps notably for the S/C/B corpus, by +11%. Russian sees big gains in all three metrics (+21% Pr, +14% Re, +19% F_1). On average, ReBPE's improvements upon BPE-knockout do cost 12 000 extra types.[4]

5.2 Ablations

We study the evolution of the tokenisers throughout the ReBPE iterations, and ablate repar-/reification (hereafter "reify") and annealing. We give a short overview of takeaways here; for numbers, see §D.

Reifying without first annealing leads to a net loss in $\lvert V\rvert$ for all but one tokeniser (see Table 5), so the +10k in Table 1 comes from annealing. Reification modifies merges much more than it creates new types (see Figure 3), and in the first iteration it even modifies more merges than knockout removes.

Reifying consistently causes a drop in F_1 right after due to worse recall, yet the knockout step that follows makes up for that entire drop *and* surpasses the gains knockout can achieve without reifying (see Table 4). Initialising the tokeniser with annealing exacerbates the drop, but the rise even more.

Finally, successive iterations of knockout mainly target merges shaped previously by knockout, rather than by reification (see Figure 4). The majority of languages show a lot of reparation (§3.1) of BPE-knockout's tuple merges, indicating high competition for tokens between merges (i.e.: if a token is no longer consumed by a merge to its left/right, a merge to its right/left eagerly awaits, which corresponds to scenario **2b** of §A.1). The S/C/B language and Polish both seem to have very little of suchlike competition, but they also have unusually small MorphyNet datasets.

[4]This is due entirely to annealing, see the ablations.

6 Releases

An implementation of ReBPE was integrated into the **BPE-knockout** Python package
`github.com/bauwenst/BPE-knockout`.
for which the HuggingFace-compatible **Tokeniser Toolkit (`TkTkT`)** package provides easy access:
`github.com/bauwenst/TkTkT`.
In addition, we release the **Morphological Decomposition and Segmentation Trove (`MoDeST`)**, a toolkit for accessing datasets like MorphyNet through a universal object-oriented interface.
`github.com/bauwenst/MoDeST`.
All tables and graphs were generated by using the **Figures as Objects (`Fiject`)** Python package:
`github.com/bauwenst/fiject`.
Lastly, the scripts needed to reproduce this paper are found at `https://github.com/bauwenst/Experiments_ReBPE`.

7 Conclusion

In this paper, we prove that an existing algorithm for aligning BPE tokenisers with desirable segmentations – BPE-knockout – does not work as intended in several ways. We propose refining it iteratively using three new mechanisms (reparation, reification, and annealing) and show that this causes Pareto-better morphological alignment F_1 (with +11% on average) in 14 European languages.

Limitations

Concatenativity As noted in the introduction, the goal of reification is to allow applying BPE-knockout several times so that blame can ripple through the BPE graph without losing other useful connection between tokens in the process. Similarly, annealing tries to find useful connections between neighbouring tokens that stay preserved across knockout. However, if tokens are related to each other *non-concatenatively*, neither of these mechanisms protect against BPE-knockout severing their connection. For example, the Arabic words for *book* and *books* are (transliterated) *kitaab* and *kutub* respectively, which follow the non-concatenative root template *k·t·b*. A triplet merge *_ku+tu+b* is doomed to be broken up further since it crosses two morpheme boundaries, despite being the only merge to keeping the three root characters together.

Vocabulary size We used the conventional (Ding et al., 2019) vocabulary size of 32ki for all tokenisers, but it is not obvious that (1) this number should still be in use today considering it originated in word-level language modelling, and that (2) it should be the same for every language considered.

Intrinsic vs. extrinsic The central hypothesis underlying all research in tokeniser alignment, e.g. by Hofmann et al. (2021) and Minixhofer et al. (2023) and Bauwens and Delobelle (2024), is that it is easier for a language model to learn a pattern optimised organically over time in a language than to learn to imitate a distortion pattern applied on top of the organic patterns by a (data-driven) tokeniser. These studies have found positive effects of (morphological) alignment on language models, so we can hypothesise that the same would hold for the present paper. However, we cannot be sure since we did not do any extrinsic evaluation.

Semi-supervision Since ReBPE uses BPE-knockout internally, it requires a dataset of desirable segmentations. The algorithm doesn't require these to be morphologically inspired, but it requires more care than scraping and processing a fully unsupervised corpus.

Acknowledgements

TB is funded by a Bijzonder Onderzoeksfonds (BOF) internal fund at KU Leuven, namely the C1 project fund with reference C14/23/096.

References

Khuyagbaatar Batsuren, Gábor Bella, and Fausto Giunchiglia. 2021. MorphyNet: a large multilingual database of derivational and inflectional morphology. In *Proceedings of the 18th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 39–48, Online. Association for Computational Linguistics.

Thomas Bauwens. 2023. BPE-knockout: Systematic review of BPE tokenisers and their flaws with application in Dutch morphology. Master's thesis, KU Leuven.

Thomas Bauwens. 2025. TkTkT: The Tokeniser Toolkit.

Thomas Bauwens and Pieter Delobelle. 2024. BPE-knockout: Pruning Pre-existing BPE Tokenisers with Backwards-compatible Morphological Semi-supervision. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 5810–5832, Mexico City, Mexico. Association for Computational Linguistics.

Pavel Chizhov, Catherine Arnett, Elizaveta Korotkova, and Ivan P. Yamshchikov. 2024. BPE Gets Picky: Efficient Vocabulary Refinement During Tokenizer Training. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 16587–16604, Miami, Florida, USA. Association for Computational Linguistics.

Marco Cognetta, Tatsuya Hiraoka, Rico Sennrich, Yuval Pinter, and Naoaki Okazaki. 2024. An Analysis of BPE Vocabulary Trimming in Neural Machine Translation. In *Proceedings of the Fifth Workshop on Insights from Negative Results in NLP*, pages 48–50, Mexico City, Mexico. Association for Computational Linguistics.

Mathias Creutz and Krista Lagus. 2007. Unsupervised models for morpheme segmentation and morphology learning. *ACM Transactions on Speech and Language Processing*, 4(1):3:1–3:34.

Pieter Delobelle, Thomas Winters, and Bettina Berendt. 2022. RobBERT-2022: Updating a Dutch Language Model to Account for Evolving Language Use. ArXiv:2211.08192 [cs].

Shuoyang Ding, Adithya Renduchintala, and Kevin Duh. 2019. A Call for Prudent Choice of Subword Merge Operations in Neural Machine Translation. In *Proceedings of Machine Translation Summit XVII: Research Track*, pages 204–213, Dublin, Ireland. European Association for Machine Translation.

Valentin Hofmann, Janet Pierrehumbert, and Hinrich Schütze. 2021. Superbizarre Is Not Superb: Derivational Morphology Improves BERT's Interpretation of Complex Words. 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 3594–3608, 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.

Mikko Kurimo, Mathias Creutz, Matti Varjokallio, Ebru Arisoy, and Murat Saraclar. 2006. Unsupervised segmentation of words into morphemes – Challenge 2005 An Introduction and Evaluation Report. In *Proceedings of the PASCAL Challenge Workshop on Unsupervised segmentation of words into morphemes*, pages 1–11.

Haoran Lian, Yizhe Xiong, Jianwei Niu, Shasha Mo, Zhenpeng Su, Zijia Lin, Hui Chen, Jungong Han, and Guiguang Ding. 2025. Scaffold-BPE: enhancing byte pair encoding for large language models with simple and effective scaffold token removal. In *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence*, volume 39 of *AAAI'25/IAAI'25/EAAI'25*, pages 24539–24548. AAAI Press.

Benjamin Minixhofer, Jonas Pfeiffer, and Ivan Vulić. 2023. CompoundPiece: Evaluating and Improving Decompounding Performance of Language Models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 343–359, Singapore. Association for Computational Linguistics.

NLLB Team, Marta R. Costa-jussà, James Cross, Onur Çelebi, Maha Elbayad, Kenneth Heafield, Kevin Heffernan, Elahe Kalbassi, Janice Lam, Daniel Licht, Jean Maillard, Anna Sun, Skyler Wang, Guillaume Wenzek, Al Youngblood, Bapi Akula, Loic Barrault, Gabriel Mejia Gonzalez, Prangthip Hansanti, John Hoffman, Semarley Jarrett, Kaushik Ram Sadagopan, Dirk Rowe, Shannon Spruit, Chau Tran, Pierre Andrews, Necip Fazil Ayan, Shruti Bhosale, Sergey Edunov, Angela Fan, Cynthia Gao, Vedanuj Goswami, Francisco Guzmán, Philipp Koehn, Alexandre Mourachko, Christophe Ropers, Safiyyah Saleem, Holger Schwenk, and Jeff Wang. 2022. No language left behind: Scaling human-centered machine translation.

Guilherme Penedo, Hynek Kydlíček, Loubna Ben allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, and Thomas Wolf. 2024. The fineweb datasets: Decanting the web for the finest text data at scale.

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.

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.

Roland Sussex and Paul Cubberley. 2006. *The Slavic languages*. Cambridge Language Surveys. Cambridge: Cambridge University Press, Cambridge, UK. Pages: xix 638.

A Proofs

A.1 Inference scenarios of BPE-knockout

We will use the Dutch example of "*bruids*-" to illustrate the effect knockout can have on inference, depending on both which merge it is applied to and what the structure of the BPE merge graph looks like. In all scenarios, we assume:

- Before knockout, the BPE tokeniser turns the string $S_1\sigma$ = "*_bruids*" into a single token [*_bruids*].
- After knockout, the unique merge that concatenated the *d* to the *s* is disabled (because the *-s* is a genitive interfix).

What we are interested in is whether all merges in the tokeniser keep being applied, whether σ = "*s*" can stay separated from S_1 = "*_bruid*" after knockout, and if so, whether S_1 and the standalone string S_2 = "*_bruid*" lead to the exact same tokens.

Scenario 1: Child knockout

If during the formation of the type *_bruids* it was the last merge that concatenated the *s* with the rest of the type, i.e. if its merge looks like $S_1 + \sigma$, then all merges that form S_1 must precede it:

```
_ + b
_b + r
_br + u
_bru + i
_brui + d

_bruid + s  // knocked
```

This means that S_1 and S_2 are merged into the same token [*_bruid*] in exactly the same way, and hence this token will be revealed when the last merge is disabled, leading to the segmentations $T(S_1\sigma)$ = [*_bruid*, *s*] and $T(S_2)$ = [*_bruid*].

Scenario 2a: Parent knockout, triplet applies

If scenario 1 does not hold, then the *d* and *s* were already concatenated at the time the final merge into *_bruids* happened. Assume for the remaining scenarios that the final merge is *_bru + ids* as in the original paper. Now, either *ids* was formed by the merge *i + ds* or by *id + s*. In scenario 2a and 2b, we assume the latter. This means *id + s* will be knocked out and the final merge *_bru + ids* will become a triplet merge *_bru + id + s*.

Since *ids* is used in the final merge, we know that the token *id* will never interact with the characters of "*_bru*" before *id+s*. However, it could do so afterwards. In scenario 2a, we assume that this is *not* the case. That is: the tokens [*id,s*] go nowhere until it is time to apply the triplet.

```
i + d
id + s  // knocked

_ + b
_b + r
_br + u

_bru + ids  // future triplet
```

This results in $T(S_1\sigma)$ = [*_bruids*] and $T(S_2)$ = [*_bru, id*], which is undesirable because no tokens are shared and a morpheme boundary is crossed. The fact that a triplet applies can hence be a good thing (*_g + id + s*) or a bad thing.

ReBPE Given that the triplet applies, it may be considered for knockout. Here, the connection between *_bru* and *id* would then be lost just to get rid of the *s*. ReBPE hence tries to add *_bru+id* into the tokeniser and replaces the triplet merge with *_bruid+s*, so that BPE-knockout's blame computation can discern which of the two is actually bad.

Scenario 2b: Parent knockout, triplet blocked

There are many possible BPE tokenisers where between the soon-to-be knocked-out merge (*id+s*) and the triplet merge (*_bru+id+s*) there exists at least one merge which either applies

(i) only to S_2 but not to $S_1\sigma$ before knockout, or

(ii) only to $S_1\sigma$ but not S_2 after knockout.

Case (i) The cause of (i) is that σ made a merge apply in $S_1\sigma$ that can't occur in S_2 (here, the merge *id+s*), meaning S_2 has an extra token (*id*) lying around that can be consumed another way. E.g.:

```
_ + b
_b + r
_br + u

i + d
id + s      // knocked

_bru + id   // only in S2
_bru + ids  // future triplet
```

Here, when knockout happens, the *id* is no longer yoinked away by *s* in $S_1\sigma$, causing *_bru+id* to apply to *both* strings. Thus, when the triplet merge

_bru+id+s is reached, $T(S_1\sigma) =$ [*_bruid, s*] and $T(S_2) =$ [*_bruid*], and therefore *the triplet is never applied, and the type _bruids can never be formed*. Notably, although there is a merge and a type in the tokeniser that can no longer occur, the two segmentations share a token, which we want.

Blocking a triplet cannot just be solved by merging some of its tokens beforehand, e.g. turning *_bru+id+s* into *_bruid+s* (or *_bru+ids*, but that means merging *id* with *s*, which was blacklisted by knockout). Consider the following tokeniser:

```
_ + b
_b + r
i + d

id + s       // knocked
u + id       // only in S2
_br + u

_bru + ids   // future triplet
```

Here, when the triplet is reached, again tokens are shared: $T(S_1\sigma) =$ [*_br, uid, s*] and $T(S_2) =$ [*_br, uid*]. So, if we want the triplet to apply, it would have to be *replaced entirely* by *_br+uid+s*.

Case (ii) Whereas case (i) was caused by σ being ignored after knockout, case (ii) is caused by σ being recruited in a merge with a token that couldn't exist before, like *uid+s*:

```
_ + b
_b + r
i + d

id + s       // knocked
u + id       // in S1 & S2 after
uid + s      // only in S1 after
_br + uid

_br + u
_bru + ids   // future triplet
```

When the triplet is reached, $T(S_1\sigma) =$ [*_br, uids*] and $T(S_2) =$ [*_bruid*]. Now the tokens are no longer shared, but still, the triplet merge *_bru+id+s* and its type *_bruids* are unreachable.

ReBPE All three tokenisers in scenario 2b have a triplet merge that is blocked from ever applying. To repair them, ReBPE applies merges to $S_1\sigma$ up to right before the triplet like above, and replaces the triplet by the actual token sequence encountered.

Scenario 3a & 3b: Grandparent knockout

Finally, if we instead assume that *d + s* and *i + ds* happen rather than *i + d* and *id + s*, then the merge to knock out becomes *d + s* and the triplet merge is *i + d + s*. Although this leaves the final merge *_bru + ids* seemingly unchanged, blocking the triplet (covered in scenario 2b) makes *_bru + ids* unreachable since it is downstream from *i+d+s*:

```
_ + b
_b + r
_br + u

d + s    // knocked
i + d
i + ds   // future triplet

_bru + ids
```

After knockout, $T(S_1\sigma) =$ [*_bru, id, s*] and $T(S_2) =$ [*_bru, id*]. The triplet *i + d + s* is never applied and neither is *_bru+ids*. Compare this to:

```
d + s    // knocked

_ + b
_b + r
_br + u

i + ds   // future triplet
i + d

_bru + ids
```

Here, the triplet is successful, so its downstream type $T(S_1\sigma) =$ [*_bruids*] can merge all the way, differing from $T(S_2) =$ [*_bru, id*].

Conclusion

Above, we have shown that there are unintended consequences to relying on tuple merges (e.g., triplets) after BPE-knockout. Except for the trivial case, the tokeniser exhibits two complementary behaviours which both defy expectations regarding its intended mechanism. In scenario 2a (noted in the original paper), the tokeniser applies the tuple merge, but for some types, this means that *despite knocking out a bad merge between characters, they still end up merged* (*ids* is no longer a type, yet *_bruids* stays reachable). In scenario 2b, BPE-knockout's tuple merges – intended to keep types in the vocabulary reachable even after losing a parent – are *blocked, and thus the vocabulary loses more types than were knocked out*. Finally, scenario 3 shows it *cascade* to distant descendants. ■

A.2 ReBPE satisfies BPE's injective variant

In proof A.1 by Bauwens and Delobelle (2024), it is shown for classical BPE that *"the same type can never be learnt by two different merges within the same vocabularisation phase"*. This then extends to inference, i.e. because only one merge will ever be learnt for the same type during training, only one merge will ever be applied to form that type during inference. This in turn naturally extends to BPE-knockout tokenisers, since their merges are a subset of a BPE tokeniser.

We now alter this proof to show a stronger statement about inference, namely that *identical tokens are merged by identical sequences of merges in any deterministic tokeniser with context-free merges*. ("Context-free" here means that deciding whether to apply a merge to a tuple of tokens is done purely based on what's *in* the tuple, not what it's surrounded by.) In other words: *even if* new merges are artificially injected into a BPE tokeniser such that the result of applying merges is no longer unique (e.g. having a tokeniser with merges *a + bcd*, *ab + cd*, and *abc + d*, all resulting in the same type *abcd*), each type will still have at most one merge that forms it, and all other merges with the same result will *never* be applied despite the tokeniser checking for their applicability.

Proof. We approach this exactly like proof A.1. Assume that we have many different strings "aSb", "cSd", ... with $S, a, b, c, d \in \Sigma^*$, all containing the same substring "S". It is given that these are all merged into the same token S. The fact that these tokens contain exactly the same characters means that no characters were taken from the surrounding strings. Since BPE merges are context-free and no characters were included from the surrounding string, the merges inside this substring happened as if there was no context at all. That is: from BPE's perspective, when it operates on "S" in "aSb", "cSd", ..., it is as if it is operating on exactly the same string "S" with no context. And since BPE is deterministic, the same string is always merged the same way, hence *no matter how many extra merges are added, and no matter how many tokens are merged by each merge*, there is only one merge that forms each type. ■

Note that you *can* change the merge tree of a type by moving merges around or inserting new merges, like ReBPE. What you *cannot* do is create more than one pathway towards a single type.

A.3 Moving merges can also block merges

Rather than injecting new merges to fill tuple merges, it is also possible to move existing merges around. We will now prove that this can in turn block other merges.

Consider the BPE tokeniser with vocabulary {*a, b, c, d, ab, abc, abcd, cd, bc, bcd*} merged as

```
a + b     // knocked
ab + c    // future triplet
abc + d

c + d
b + c     // future submerge
b + cd
```

Knocking out the type *ab* gives

```
a + b + c
abc + d

c + d
b + c
b + cd
```

We now want to turn the triplet merge back into a binary merge by adding the submerge *b + c* before it. Since this is an existing merge in the tokeniser, we have to instead move merges around in order to make *a + bc* an applicable merge. We have two options: firstly, we could delay the triplet so that it appears only once *bc* has formed.

```
abc + d   // blocked

c + d
b + c
a + bc    // moved
b + cd
```

This clearly makes the first merge impossible, because *abc* is needed sooner than it is formed. Secondly, we could expedite the submerge so that it happens before the triplet:

```
b + c   // moved
a + bc
abc + d

c + d
b + cd  // blocked
```

Here too, a merge is now blocked. The reason is that *b + c* steals the *c* necessary to apply *c + d*. We could in turn expedite *c + d*:

```
c + d  // moved
b + c  // moved
a + bc
abc + d  // blocked

b + cd
```

... and now indeed, another merge is again blocked because *c + d* steals the *c* necessary for *b + c*, the exact converse of the problem we just resolved.

A.4 Relationship between ScaffoldBPE and PickyBPE criteria

Let f' measure the frequency of types and pairs of types in a corpus before applying merge (x, y), and let f measure frequencies after doing so.

The criterion for marking a type x as scaffolding in ScaffoldBPE is

$$f(x) \leq f(a, b) \tag{1}$$

where (a, b) is the next pair to merge. This happens when x has become so depleted due to the merge that it is, across all remaining pairs it appears in, less frequent than just one pair whose frequency is already below that of (x, y).

In PickyBPE, the criterion is

$$\frac{f'(x, y)}{f'(x)} \geq \mathcal{T} \tag{2}$$

for some $\mathcal{T} \in [0, 1]$, and practically $\mathcal{T} \in [0.5, 1]$. This happens when a large enough part of x's occurrences before merging are actually next to y.

A.4.1 PickyBPE as ScaffoldBPE

Since ScaffoldBPE is phrased in terms of the present, we do the same for PickyBPE now.

Since $f(x) = f'(x) - f'(x, y)$, we can write

$$\begin{aligned} \frac{f'(x, y)}{f'(x)} &\geq \mathcal{T} \\ f'(x, y) &\geq \mathcal{T} \cdot f'(x) \\ f'(x, y) &\geq \mathcal{T} \cdot (f(x) + f'(x, y)) \\ (1 - \mathcal{T}) \cdot f'(x, y) &\geq \mathcal{T} \cdot f(x) \end{aligned} \tag{3}$$

and thus

$$f(x) \leq \frac{1 - \mathcal{T}}{\mathcal{T}} \cdot f'(x, y) = \lambda \cdot f(xy) \tag{4}$$

where $\lambda \in [0, 1]$ assuming $\mathcal{T} \in [0.5, 1]$. ■

A.4.2 ScaffoldBPE as PickyBPE

Conversely, PickyBPE is phrased in terms of the past. We get

$$\begin{aligned} f(x) &\leq f(a, b) \\ f'(x) - f'(x, y) &\leq f(a, b) \\ f'(x) - f(a, b) &\leq f'(x, y) \end{aligned} \tag{5}$$

and thus

$$\frac{f'(x, y)}{f'(x)} \geq 1 - \frac{f(a, b)}{f'(x)} = \mathcal{T}(x, a, b) \tag{6}$$

where $\mathcal{T}(x, a, b) \in [0, 1]$ because of the transitivity of $f'(x) \geq f'(x, y) \geq f(a, b) > 0$. ■

B Experimental Setup

B.1 Data processing

Matching sizes To match the "amount of language" we consider from each corpus, we first compute by which factor c_i the amount of characters used in another language i exceeds the amount used to express the same statements in English in the multi-parallel FLORES-200 corpus (NLLB Team et al., 2022). Then, we measure the amount of characters n in the first 30 million English Fineweb examples, and take from the other (non-parallel) corpora i as many examples as needed to reach $n \times c_i$ characters. By using characters rather than e.g. bytes, we avoid any confounding artifacts introduced by an arbitrary choice of encoding algorithm (e.g. UTF-8) to needlessly serialise the given text to a format no human can read anyway.

String filtering ($\mathcal{F}_1$) Because BPE is (at least in this paper) applied within words independent of context, we convert each matched corpus into a word frequency list using a simple preprocessor:

1. Apply NFKC normalisation.
2. Split the result on punctuation, except for (1) hyphens and (2) apostrophes surrounded by only non-space.
3. Isolate groups of the same punctuation mark within the resulting strings that are pure punctuation.
4. Split the resulting strings on whitespace without keeping the whitespace characters.

The remaining strings are counted, and afterwards, all strings with a frequency under 5 are considered spurious and therefore removed. This cuts down on more than half of the words to store and preprocess whilst preserving well over 95% of the data.

Character filtering ($\mathcal{F}_2$) For training BPE, we use four nines of character coverage, i.e. 0.9999 of the non-unique characters in the corpus are preserved and the rest (starting with the least frequent characters) are replaced by `[UNK]`. We found 0.9999 to produce a good trade-off between capturing all the important characters (including punctuation) whilst avoiding thousands of emoji and foreign scripts – despite Fineweb's supposed cleanliness. We ensure the ASCII range $\{33, ..., 122\}$ is in the vocabulary by adding any missing characters after character coverage.

Preprocessing When preprocessing a word for vocabularisation or segmentation, a more elaborate pipeline is used:

1. Apply NFKC normalisation.
2. Split the result on punctuation, except for (1) hyphens and (2) apostrophes surrounded by only non-space.
3. Split the resulting strings on whitespace without keeping the whitespace characters.
4. Split any English contractions off the resulting strings ('ve, 'll, ...) by explicitly matching against them.
5. Find apostrophes surrounded by non-spaces, split on them, and stick each apostrophe to whichever neighbouring string is longer. (This is an approximation of the previous step for contractions in other languages, e.g. the French *l'* or the Catalan *'ns*.)
6. Add a prefix character _ to the resulting strings.
7. Group digits in runs of three in the resulting strings, starting from the right.
8. Isolate (runs of) hyphens that have a character to the left and the right in the resulting strings.
9. Lowercase the resulting strings, but do it losslessly by adding a capitalising token as the first pretoken of the string, if the first non-prefix character is a capital.[5]

[5]E.g.: the string _Шрёдингер would become the strings [⇑, _шрёдингер].

B.2 Dataset sizes

As mentioned, we take the first 30 million examples in the English Fineweb corpus and count how many characters it contains, which is then multiplied by each language's FLORES-200 character premium to obtain a target amount of characters. This target amount is used to truncate the other Fineweb corpora, which are then processed as per above. Table 2 displays this target amount, as well as the amount of characters that remain after $\mathcal{F}_1$. These are what the vocabulariser sees.

The MorphyNet sizes are in Table 3.

	CP	Chars taken	Chars after $\mathcal{F}_1$
Catalan	1.104	98 383 288 869	34 874 706 586
Czech	0.969	86 339 772 476	72 579 997 825
English	1.000	89 143 444 024	73 567 572 983
Finnish	1.066	95 002 798 645	81 977 270 350
French	1.192	106 221 644 586	88 572 587 933
German	1.168	104 120 970 800	87 921 045 430
Hungarian	1.052	93 818 656 458	80 053 960 264
Italian	1.179	105 112 724 547	88 392 589 424
Polish	1.061	94 618 021 763	80 750 561 058
Portuguese	1.087	96 862 206 926	80 670 417 842
Russian	1.090	97 129 470 868	83 080 371 759
S/C/B	0.993	88 493 483 075	74 127 746 557
Spanish	1.191	106 137 409 647	88 220 679 849
Swedish	1.006	89 662 026 199	74 716 740 653

Table 2 – Character amounts for training the tokenisers on Fineweb. "CP" is character premium.

	Total	=	Infl.	+	Deriv.
Catalan	133 384	=	125 366	+	8018
Czech	351 298	=	318 962	+	32 336
English	416 202	=	191 071	+	225 131
Finnish	1 677 664	=	1 640 821	+	36 843
French	423 724	=	350 772	+	72 952
German	211 287	=	181 906	+	29 381
Hungarian	952 408	=	924 232	+	28 176
Italian	629 276	=	570 428	+	58 848
Polish	58 711	=	0	+	58 711
Portuguese	288 266	=	276 492	+	11 774
Russian	838 539	=	745 500	+	93 039
S/C/B	2876	=	0	+	2876
Spanish	939 284	=	908 507	+	30 777
Swedish	103 899	=	94 655	+	9244

Table 3 – Amount of examples available in MorphyNet. We take a 250 000-example sample without replacement, in those cases said limit is exceeded.

B.3 Hardware

The corpora were preprocessed on 14 parallel cores of an Intel Xeon Platinum 8360Y CPU (2.4 GHz) with ∼28 GiB of RAM each, totalling the equivalent of a little over 6 CPU days. The parallel runs

all finished within 16 hours.

Training the baseline, vanilla BPE tokenisers was done on the same CPU setup but with an expanded ∼100 GiB of RAM each, totalling 6 CPU hours (the longest run finishing after 1.5 hours).

"Fine-tuning" (i.e. applying the ReBPE algorithm) over the 250k MorphyNet sample was done on 14 parallel cores of an Intel Core Ultra 7 155H laptop CPU with ∼32 GiB of RAM shared across the cores, finishing after 2.5 hours (so, somewhere less than 35 CPU hours total).

Evaluation on the same setup took 20 minutes (so less than 5 CPU hours).

We also found that these resources were sufficient for training the baseline tokenisers in most languages of a few million unique words, provided the memory was fully allocated to one language at a time.

C Algorithms

Below are pseudocode implementations for the algorithms of §3. They are explained further there. One notable detail is what happens when a binary merge (p_1, p_2) is reified whose result p_1p_2 is not part of the vocabulary (when the check on line 32 is successful): in that case, a new merge will be created which occurs *before* any of the tuple merges to which it is subsequently applied but *after* all merges that happen before those. This is done using a fractional rank that is a small offset ε smaller than the lowest rank of the affected tuple merges.

In these implementations, function calls with * have the structure of the tokeniser $(V, \mathcal{M}_i, \mathcal{M}_o)$ as an implicit input and an implicit output. (Indeed, procedural pseudocode is notationally quite heavy without introducing what are essentially object-oriented fields and methods.)

Variables ending in a question mark are boolean and are used to configure ablations.

Algorithm 1 ReBPE: iterative BPE-knockout with reification in between.

```
1  function REBPE(V, M_i, M_o, I)
2      if anneal? then
3          ANNEALPHASE*(D)
4      finished ← I = 0
5      blacklist ← {}
6      for i ∈ 1, ..., I do
7          removed ← KNOCKOUTPHASE*(D)
8          finished ← true
9          blacklist ← blacklist ∪ removed
10         if not reify? then
11             continue
12         added ← REIFYPHASE*(blacklist)
13         finished ← |added| = 0
14         if |removed ∪ added| = 0 then
15             break
16     if not finished then
17         KNOCKOUTPHASE*(D)
18     return (V, M_i, M_o)
```

Algorithm 2 Reification of a BTE tokeniser

```
1  function REIFYPHASE(V, M_i, M_o, ~~M~~)
2      for s, m ∈ ENTRIES(M_i) do
3          if not fix? then
4              break
5          if |PARENTS(m)| < 3 then
6              continue
7          tokens ← LIMITEDBPE*(s, RANK(m))
8          if tokens ≠ PARENTS(m) then
9              for p ∈ PARENTS(m) do
10                 M_o(p) ← M_o(p) \ {m}
11             m ← (RANK(m), tokens)
12             for p ∈ PARENTS(m) do
13                 M_o(p) ← M_o(p) ∪ {m}
14             M_i(s) ← m
15             applied ← applied ∪ {m}
16     if not link? and not expand? then
17         return (V, M_i, M_o), applied
18     new ← MAP({})
19     for ·, m ∈ ENTRIES(M_i) do
20         p_1 ... p_n ← PARENTS(m)
21         if n = 2 then
22             continue
23         for i ∈ 1 ... n − 1 do
24             m_new ← (p_i, p_{i+1})
25             if m_new ∉ ~~M~~ then
26                 new(m_new) ← new(m_new) ∪ {m}
27     applied ← {}
28     for (p_1, p_2), tuples ∈ ENTRIES(new) do
29         tuples ← {t ∈ tuples | FIND((p_1, p_2), t)}
30         if |tuples| = 0 then
31             continue
32         if p_1p_2 ∉ V then
33             if not expand? then
34                 continue
35             k ← RANK(min(tuples)) − ε
36             m ← (k, (p_1, p_2))
37             V ← V ∪ {p_1p_2}
38             M_i(p_1p_2) ← m
39             for i ∈ 1 ... 2 do
40                 M_o(p_i) ← M_o(p_i) ∪ {m}
41         else
42             m ← M_i(p_1p_2)
43         for t ∈ tuples do
44             if RANK(m) ≥ RANK(t) then
45                 continue
46             for p ∈ PARENTS(t) do
47                 M_o(p) ← M_o(p) \ {t}
48             t ← REPLACE((p_1, p_2), p_1p_2, t)
49             for p ∈ PARENTS(t) do
50                 M_o(p) ← M_o(p) ∪ {t}
51             M_i(CHILD(t)) ← t
52             applied ← applied ∪ {m}
53     return (V, M_i, M_o), applied
```

D Ablations and Iteration Diagnostics

	anneal	reify	Iteration 0 BPE	Iteration 0 +anneal	Iteration 1 +knock	Iteration 1 +reify	Iteration 2 +knock	Iteration 2 +reify	Iteration 3 +knock	Iteration 3 +reify	Iteration 4 +knock	Iteration 4 +reify	Iteration 5 +knock	Iteration 5 +reify	Iteration 6 +knock	Iteration 6 +reify	Iteration 7 +knock	Iteration 7 +reify	F_1
Catalan	×	×	23.07		+29.33		+2.31		+0.12		+0.00								54.84
	×	✓			+29.33	−5.02	+8.55	−0.65	+2.50	+0.04	+0.25	+0.01	+0.05	+0.00					58.13
	✓	×		+4.67	+31.02		+3.62		+0.52		+0.01								62.91
	✓	✓		+4.67	+31.02	−9.56	+14.33	−1.87	+4.08	−0.10	+0.30	+0.00	+0.01	+0.00					**65.95**
Czech	×	×	10.33		+22.28		+2.96		+0.21										36.81
	×	✓			+22.28	−5.41	+9.53	−1.30	+1.46	−0.00	+0.36	−0.04	+0.04	−0.01	+0.01	+0.00			38.34
	✓	×		+2.13	+24.05		+4.41		+0.35		+0.02								42.70
	✓	✓		+2.13	+24.05	−8.71	+14.30	−2.42	+2.97	−0.04	+0.55	−0.14	+0.14	−0.00	+0.02	−0.00			**44.70**
English	×	×	19.11		+21.07		+0.39												40.89
	×	✓			+21.07	−7.15	+7.56	−0.63	+1.22	−0.10	+0.11	−0.00	+0.01	+0.00					41.52
	✓	×		+3.91	+21.95		+1.48												46.86
	✓	✓		+3.91	+21.95	−10.58	+11.05	−1.67	+3.49	−0.18	+0.32	+0.00	+0.02	−0.00					**47.86**
Finnish	×	×	14.64		+13.76		+0.98		+0.00										29.44
	×	✓			+13.76	−0.80	+2.25	−0.08	+0.20	+0.00	+0.01	+0.00							30.02
	✓	×		+3.15	+15.50		+1.85		+0.02										35.22
	✓	✓		+3.15	+15.50	−3.22	+5.37	−0.50	+0.73	−0.01	+0.05	+0.00	+0.00	+0.00					**35.78**
French	×	×	24.82		+15.36		+0.56		+0.00										40.99
	×	✓			+15.36	−1.57	+3.26	−0.16	+0.33	+0.27	+0.01	+0.00							42.59
	✓	×		+6.93	+19.31		+0.94		+0.01										52.42
	✓	✓		+6.93	+19.31	−6.57	+7.87	−1.12	+1.40	−0.02	+0.03	+0.00							**53.07**
German	×	×	13.98		+9.08		+0.46												28.28
	×	✓			+9.08	−1.48	+2.09	−0.22	+0.28	−0.02	+0.02	+0.00							28.56
	✓	×		+2.43	+9.69		+1.25												34.13
	✓	✓		+2.43	+9.69	−2.91	+3.98	−0.34	+0.54	−0.02	+0.09	+0.00							**34.31**
Hungarian	×	×	41.60		+16.30		+0.78		+0.01										59.46
	×	✓			+16.30	−1.53	+3.07	−0.88	+1.20	−0.01	+0.01	+0.00	+0.00	+0.00					60.56
	✓	×		+10.18	+18.54		+1.33		+0.04										72.85
	✓	✓		+10.18	+18.54	−4.13	+5.77	−1.04	+1.63	−0.05	+0.15	−0.00	+0.00	+0.00					**73.85**
Italian	×	×	11.57		+14.07		+0.40												26.05
	×	✓			+14.07	−2.90	+3.97	−0.05	+0.68	+0.01	+0.18	−0.02	+0.02	−0.00	+0.01	−0.00			27.54
	✓	×		+3.34	+15.75		+1.63		+0.01										32.31
	✓	✓		+3.34	+15.75	−7.10	+9.25	−0.61	+1.40	+0.00	+0.22	−0.00	+0.01	−0.00	+0.01	−0.00			**33.86**
Polish	×	×	18.46		+17.93		+0.56		+0.00										37.70
	×	✓			+17.93	−2.24	+4.16	−0.45	+0.78	+0.01	+0.02	+0.00							39.45
	✓	×		+3.09	+17.53		+2.40		+0.01										42.44
	✓	✓		+3.09	+17.53	−4.15	+8.08	−0.83	+1.57	−0.01	+0.09	+0.00							**44.86**
Portuguese	×	×	22.78		+20.39		+0.98		+0.01										44.18
	×	✓			+20.39	−0.43	+3.82	+0.11	+0.40	+0.00	+0.02	+0.00							47.11
	✓	×		+7.10	+26.32		+1.94		+0.02										58.20
	✓	✓		+7.10	+26.32	−9.91	+12.48	−0.83	+1.21	+0.03	+0.04	+0.00							**59.26**
Russian	×	×	16.73		+18.08		+4.17		+0.05		+0.00								39.04
	×	✓			+18.08	−3.44	+13.00	−6.37	+9.00	−1.86	+1.99	−0.76	+0.97	+0.00	+0.68	−1.12	+1.16	+0.00	48.08
	✓	×		+3.44	+19.33		+5.19		+0.25		+0.52								45.48
	✓	✓		+3.44	+19.33	−8.23	+16.03	−7.47	+10.96	−2.21	+3.61	−1.11	+1.31	−0.01	+0.92	−1.29	+1.29	+0.00	**53.34**
S/C/B	×	×	19.75		+28.74		+4.00												52.90
	×	✓			+28.74	+0.36	+7.34	−0.30	+0.90	+0.06	+0.17	+0.00							57.51
	✓	×		+0.47	+29.49		+4.11												54.26
	✓	✓		+0.47	+29.49	+0.13	+7.45	−0.28	+0.88	+0.06	+0.17	+0.00							**58.63**
Spanish	×	×	34.57		+15.27		+0.28		+0.00										50.13
	×	✓			+15.27	−0.98	+2.12	−0.08	+0.23	+0.00	+0.01	+0.00							51.14
	✓	×		+8.91	+16.05		+1.84		+0.03										61.40
	✓	✓		+8.91	+16.05	−4.46	+7.01	−0.64	+1.02	−0.15	+0.15	+0.00	+0.02	−0.00					**62.48**
Swedish	×	×	23.81		+33.50		+1.37		+0.04		+0.01								58.76
	×	✓			+33.50	−8.71	+10.63	−0.62	+1.07	−0.02	+0.13	+0.00	+0.01	+0.00					59.83
	✓	×		+4.86	+36.67		+1.89		+0.18		+0.01								67.46
	✓	✓		+4.86	+36.67	−12.97	+15.74	−1.20	+2.33	−0.05	+0.27	+0.00	+0.01	+0.00	+0.00	+0.00			**69.53**

Table 4 – Evolution of the morphological micro-F_1 score throughout iterative application of knockout and optionally reification, optionally preceded by annealing, starting from the 32ki BPE tokenisers of Table 1. Note how reification causes bigger knockout gains and annealing causes bigger reification losses.

	× / ×	× / ✓	✓ / ×	✓ / ✓
Catalan	27 885	28 193	35 484	34 653
Czech	29 463	29 660	46 135	45 925
English	29 721	29 227	46 223	44 508
Finnish	31 234	32 332	48 075	48 535
French	28 924	28 722	45 831	43 252
German	30 448	30 715	47 397	47 570
Hungarian	29 557	29 743	46 497	45 761
Italian	31 048	32 447	47 856	48 957
Polish	31 083	32 271	37 641	38 876
Portuguese	28 377	29 181	44 860	42 082
Russian	31 073	32 104	47 847	48 099
S/C/B	31 545	34 230	31 587	34 265
Spanish	30 127	31 552	46 672	47 368
Swedish	29 366	28 836	39 523	38 169

Table 5 – Vocabulary size $|V|$ after the last iteration for each of the tokenisers in Table 4.

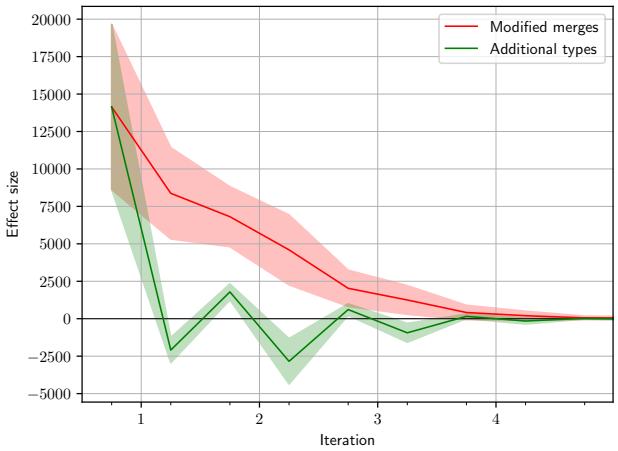
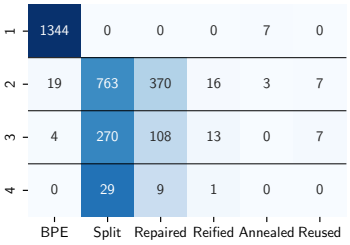
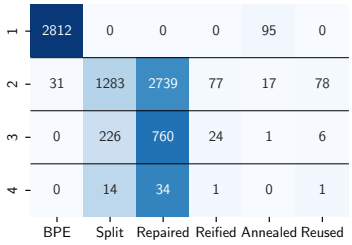


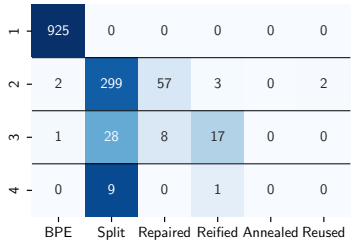
Figure 3 – Mean and std across languages of the amount of changes to the vocabulary and merges throughout iterations.



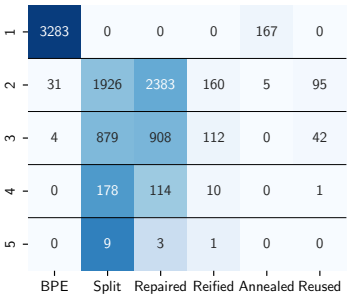
(a) Polish



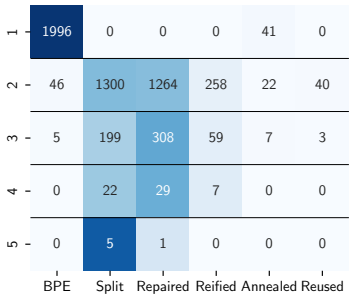
(b) French



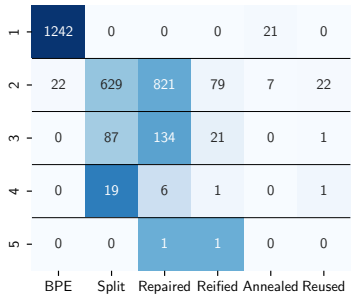
(c) S/C/B



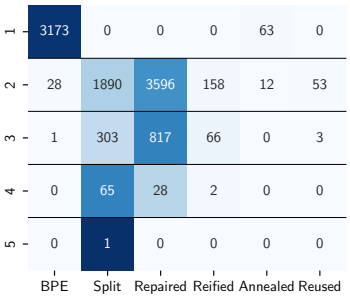
(d) Catalan



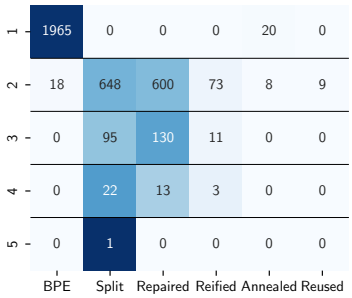
(e) Spanish



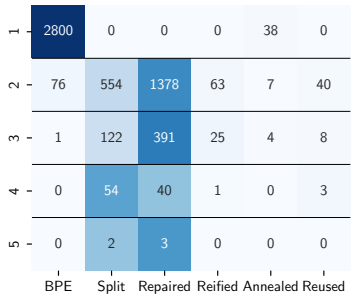
(f) Finnish



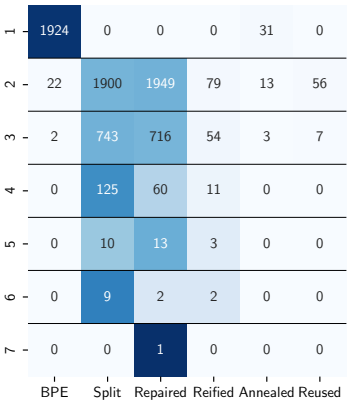
(g) Portuguese



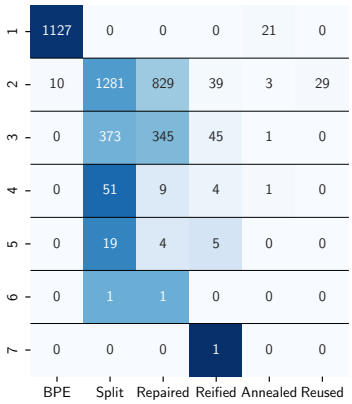
(h) German



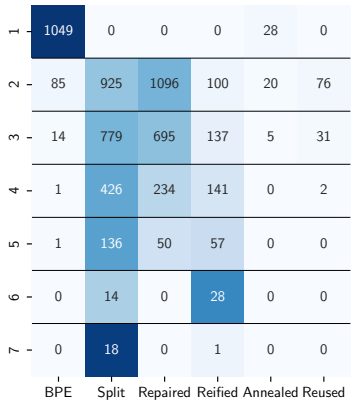
(i) Hungarian



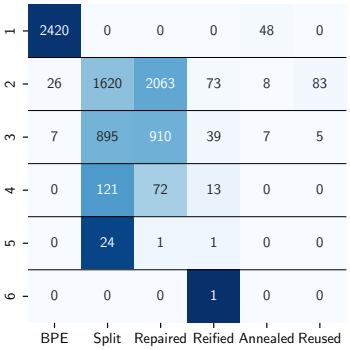
(j) Czech



(k) Italian



(l) Russian



(m) English

(n) Swedish

Figure 4 – Amount of merges knocked out per iteration of knockout in each ReBPE tokeniser (i.e. with annealing before and reification between), stratified by how the merges came into existence before. "Split" means a merge was made when knockout split one of the tokens of an existing merge into two parts, making that merge a tuple merge (it is not necessarily a tuple merge when it is knocked out since reification could have contracted the rest of the tuple). "Reified" denotes a binary merge created by reification. "Reused" denotes a binary merge used in some preceding reification step, at which point it somehow already existed.