# CodeBPE: Investigating Subtokenization Options for Large Language Model Pretraining on Source Code

**Anonymous ACL submission**

## Abstract

Recent works has widely adopted large language model pretraining for source code, suggested source code-specific pretraining objectives and investigated the applicability of various Transformer-based language model architectures for source code. This work investigates another important aspect of such models, the effect of different subtokenization options, and aims at identifying most effective and length-efficient subtokenizations, taking into account source code specifics. We propose subtokenziation that reduces average length by 17–40% without downstream performance drop, and show that a carefully chosen subtokenization may significantly improve quality by 0.5-2%, possibly with some length increase.

## 1 Introduction

With the inspiration from the success of large language model (LM) pretraining in natural language processing (NLP), BERT-like models have been widely adopted for source code processing (Feng et al., 2020; Kanade et al., 2020), as code has similar discrete sequential structure to natural text. Being trained on huge source code corpora in a self-supervised manner, large LMs often substantially outperform domain-specific models developed purposely for applied tasks, especially in the tasks with limited parallel / labelled data (Ahmad et al., 2021a). These tasks include fixing code bugs, generating text from code and vice versa, or translating code from one programming language to another.

Recent works advanced large LM pretraining on source code in two main directions. First, various model kinds were utilized for source code: CodeBERT (Feng et al., 2020) and CuBERT (Kanade et al., 2020) rely on the classic encoder-only RoBERTa (Liu et al., 2019), CodeGPT (Lu et al., 2021) uses decoder-only GPT (Radford and Narasimhan, 2018), PLBART (Ahmad et al., 2021a) is based on the denoising sequence-to-sequence BART (Lewis et al., 2020) model, and

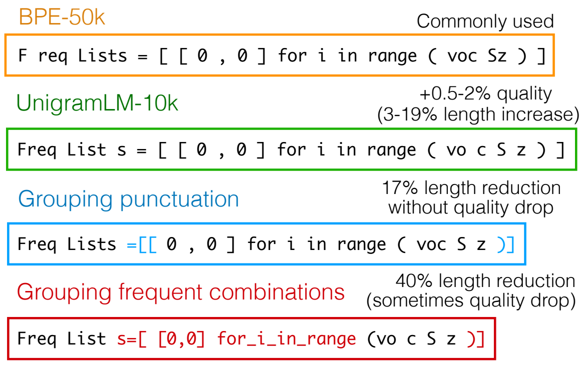


Figure 1: Example subtokenizations (all numbers compared to the commonly used BPE-50k).

CodeT5 (Wang et al., 2021b) utilizes multitask sequence-to-sequence T5 (Raffel et al., 2020). Second, a range of code-specific self-supervised pretraining tasks were proposed to enrich the classic masked language modeling (MLM) objective, e. g. GraphCodeBERT (Guo et al., 2021) predicts data flow connections during pretraining (one variable is computed from another variable), and CodeT5 (Wang et al., 2021b) and DOBF (Roziere et al., 2021) use a variable naming objective.

This work is devoted to investigating one more important component which is usually not paid much attention when pretraining large LMs on source code — subtokenization. Modern LMs usually preprocess sequences using open-vocabulary models such as Byte-pair encoding (BPE) which splits long tokens into smaller subtokens, in order to ensure the relatively high frequency of all subtokens. Though this process is often referred to as tokenization, we call it subtokenization, to underline its smaller granularity.

Though subtokenization is often chosen with only superficial deliberation, it is one of the essential model components which may affect both quality and prediction speed. First, an inaccurately chosen subtokenization procedure may substantially increase sequence lengths and consequently slow down prediction. As a simple example, the work on CodeT5 (Wang et al., 2021b) notices that using BPE trained specifically on source code corpora makes sequences 30–45% shorter than using BPE trained on natural text. Second, a line of works indicates the positive effect of the carefully chosen subtokenization procedure on the model effectiveness in NLP. For example, Bostrom and Durrett (2020) show that using a UnigramLM (Kudo, 2018) algorithm for subtokenization instead of BPE improves the quality of BERT-based question answering or textual entailment in English by 1%, and Ding et al. (2019) show that tuning BPE vocabulary size in machine translation may produce +4 BLEU. At the same time, for large LMs, the particular subtokenization procedure chosen at the pretraining stage becomes an inseparable part of the model and must later be used in applied tasks. This underlines the need for a careful choice of subtokenization options when pretraining large LMs.

In this work, we conduct a deep study of subtokenization options for large LM pretraining on source code, using PLBART as a testing ground. In addition to investigating general aspects, e. g. the subtokenization algorithm and the vocabulary size, we study the ways of adapting subtokenization to the specific properties of code, such as a large amount of punctuation marks and frequently-used token combinations, a variety of complex identifiers (e. g. variable or function names), or relative similarity of programming languages. We aim at choosing optimal subtokenization options that (a) lead to the best performance or (b) minimize sequence lengths (and thus speed up the model) without downstream performance drop. Our contributions are as follows - we show that for large LMs pretrained on source code:

- grouping punctuation chars in single tokens reduces the average length by 17% without downstream performance drop, and allowing more complex composite tokens reduces lengths by 40%, sometimes with quality drop;
- UnigramLM is generally preferable over BPE;
- smaller vocabularies may improve downstream quality with 3–19% length increase;
- subtokenizers are well transferable between programming languages;
- BPE-dropout (Provilkov et al., 2020) may improve quality in tasks with small data.

Our length-efficient subtokenization procedure (see examples in Figure 1) compresses sequences by 17–40% without quality drop and our most effective subtokenization improves quality by 0.5–2% significantly in three out of eight tasks and by one standard deviation – in other three tasks.

## 2 Methodology and experimental setup

The existing works on large LMs for source code usually choose a particular subtokenization library, for example the same as in the base LM the work uses, and train the subtokenizer with the vocabulary size of 30-50K on source code corpora used for pretraining. Often code is preprocessed before subtokenization, e. g. by replacing `\n` with `NEW_LINE`, and split into tokens on white-spaces and punctuation marks so that these tokens are further split into subtokens, e. g. `for i in range (vocSize)` will be split into [‘`for`’, ‘`i`’, ‘`in`’, ‘`range`’, ‘`(`’, ‘`vocSize`’, ‘`)`’] even if `for i in` is generally a frequent combination. The latter principle appears to be intuitively reasonable, since it ensures that subtokenization preserves syntactically meaningful boundaries of tokens (Kanade et al., 2020). We refer to this principle as prohibiting *composite tokens*. More details on subtokenization in different pretrained LMs for code are given in Section 8.

We treat the described commonly-used approach as a baseline, and conduct a series of experiments, each modifying the baseline subtokenization procedure in one dimension, e. g. changing the subtokenization algorithm, and pretraining PLBART with the new subtokenization. As a baseline, we use a (slightly modified, see details below) subtokenization procedure of PLBART. The dimensions we vary are as follows: the subtokenization algorithm, restrictions on preliminary splitting, the vocabulary size, the set of languages the subtokenizer is trained on, and the use of stochastic subtokenization. These dimensions are inspired either by the specifics of source code or by recent works on subtokenization in NLP.

**Experimental setup.** As our base model, we use PLBART (Ahmad et al., 2021a), since it comes with the released pretraining code and data preprocessing routine. We use the same model size, the pretraining dataset size and other hyperparameter settings, including finetuning hyperparameters, as in PLBART. Particularly, we use an encoder-decoder Transformer architecture with 6 layers in each part, with the model dimension of 768 and

12 heads (140M parameters). The pretraining data consists of 230M Python functions, 470M Java functions and 47M natural language (NL) descriptions, called sequences below.

We pretrain all our PLBART models for 100k updates, as in the original paper. We clip all sequences by 510 subtokens, which remains the majority (96-99.1%) of sequences unclipped in all subtokenizations. The average length reported in the paper is computed on the randomly chosen subset of pretraining data before clipping.

As applied tasks, we consider three tasks from the PLBART paper: code generation (generating a Java function based on an NL description; CONCODE (Iyer et al., 2018) dataset, CodeBLEU (Ren et al., 2020) metric), code summarization (generating an NL description for a Python or Java function; CodeSearchNet (Husain et al., 2020) dataset, BLEU metric), code clone detection (classifying whether two Java functions implement the same functionality; BigCloneBench dataset (Svajlenko and Roy, 2015); F1 metric), and one additional task of code translation (translating code from Python to Java and vice versa; AVATAR dataset (Ahmad et al., 2021b)). Here we consider original data with the CodeBLEU metric (Code Translation-1) and the smaller version of data with tests and the Computational Accuracy metric – which portion of generated functions passed all tests (Code Translation-2). We chose tasks so that we have both code generative and discriminative tasks and that datasets are in Python or Java.

**Baseline subtokenization.** Following Ahmad et al. (2021a), we use a SentencePiece (Kudo and Richardson, 2018) library, which is today one of the most widely used solutions for subtokenization. We train subtokenizers on 10M functions and NL descriptions randomly selected from the pretraining data. Though Ahmad et al. (2021a) use BPE subtokenization algorithm, our baseline subtokenization uses another algorithm, UnigramLM, because it was shown to be quantitetively and qualitatively more suitable for pretraining in NLP than BPE. We also perform their comparison for code in Section 4. We set the vocabulary size to 50K (the commonly used size for large LMs for code) and character coverage to 99.99% (enough to cover English chars and punctuation).

We also use PLBART's preprocessing which includes removing comments and docstrings, replacing `\n`, indents and dedents in Python with `NEW_LINE`, `INDENT` and `DEDENT` tokens as they are a part of the language syntax, and removing formatting in Java as it does not affect the language syntax. Our baseline subtokenizer follows the commonly used strategy of prohibiting composite tokens described above. The only exception we make is that we do not split identifiers by underscores _ because they do not represent a syntax unit, as other punctuation chars do.

## 3 Subtokenization granularity

In contrast to natural text in which a portion of punctuation chars is small and thus their separation in subtokenization does not affect lengths much, in source code, punctuation constitutes 12.8% of chars and often forms frequent combinations joining which into composite tokens may substantially reduce lengths. Further, the presence of a large amount of commonly used patterns is another specific feature of source code, e. g. `for (int i = 0;` in Java or `def __init__ (self):` in Python, and these patterns again may form composite tokens. This section investigates the effects of the use of composite tokens on performance and length-efficiency.

We consider several levels of allowed complexity of composite tokens listed in Table 1 and empirically compare them in Figure 2. The two extreme cases are no composite tokens (Level 0, equal to baseline tokenization) and unrestricted composite tokens complexity (Level 4, composite tokens constitute 48.6% of the vocabulary). The average sequence length in Level 4 is 40% less than that in Level 0. At the same time, the effect on performance depends on the task: in code-generative tasks (translation and generation), Level 4 performs significantly worse than Level 0, and in code understanding tasks, Level 4 is either similar / marginally worse than Level 0 (code summarization) or even significantly better (clone detection). Because of quality loss encountered in several tasks, we consider intermediate levels.

Level 1 makes one step further from Level 0 and allows punctuation char merges, e. g. '`})`' or '`]):`'. Though such punctuation composite tokens only occupy 3.4% of the vocabulary, their use reduces average length by 17%: from 97 to 80.7, and since this level does not mix punctuation with other chars, it presumably should not complicate code processing much. Level 2 makes one more step further and allows merging dots `.` with textual tokens.

| Level | Description | Example |
|---|---|---|
| 0 | Whitespaces in the middle of tokens are prohibited and each punctuation char is treated as a separate token (except '_') | [‘for’, ‘i’, ‘in’, ‘range’, ‘(’, ‘df’, ‘.’, ‘shape’, ‘[’, ‘1’, ‘]’, ‘)’, ‘:’, ‘NEW_LINE’, ‘INDENT’, ‘print’, ‘(’, ‘i’, ‘)’, ‘NEW_LINE’, ‘print’, ‘(’, ‘df’, ‘.’, ‘columns’, ‘[’, ‘i’, ‘]’, ‘)’] |
| 1 | Similar to Level 0, but tokens consisting of several punctuation chars are allowed | [‘for’, ‘i’, ‘in’, ‘range’, ‘(’, ‘df’, ‘.’, ‘shape’, ‘[’, ‘1’, ‘**] ) :**’, ‘**NEW_LINE INDENT**’, ‘print’, ‘(’, ‘i’, ‘**) NEW_LINE**’, ‘print’, ‘(’, ‘df’, ‘.’, ‘columns’, ‘[’, ‘i’, ‘**] )**’] |
| 2 | Similar to Level 1, but dots are allowed in tokens | [‘for’, ‘i’, ‘in’, ‘range’, ‘(’, ‘df’, ‘**.shape**’, ‘[’, ‘1’, ‘**] ) :**’, ‘**NEW_LINE INDENT**’, ‘print’, ‘(’, ‘i’, ‘**) NEW_LINE**’, ‘print’, ‘(’, ‘df’, ‘**.columns**’, ‘[’, ‘i’, ‘**] )**’] |
| 3 | Whitespaces and single punctuation chars allowed in tokens, except NEW_LINE | [‘**for i in range**’, ‘**( df**’, ‘**. shape [ 1**’, ‘**] ) :**’, ‘**NEW_LINE INDENT**’, ‘print’, ‘**( i**’, ‘**) NEW_LINE**’, ‘print’, ‘**( df**’, ‘**. column**’, ‘**s [ i**’, ‘**] )**’] |
| 4 | Composite tokens of arbitrary complexity are allowed | [‘**for i in range**’, ‘**( df**’, ‘**. shape**’, ‘**[ 1 ]**’, ‘)’, ‘**: NEW_LINE**’, ‘**INDENT print**’, ‘( i )’, ‘**NEW_LINE print**’, ‘**( df**’, ‘**. columns**’, ‘**[ i ] )**’] |

Table 1: Different levels of allowed composite tokens complexity considered in the paper. **Green** emphasizes tokens which could not be obtained in the previous level, and **gray** emphasises the remaining tokens that could not be obtained in Level 0. Levels list *allowed* merges, but what particular merges to perform is chosen by the tokenizer.

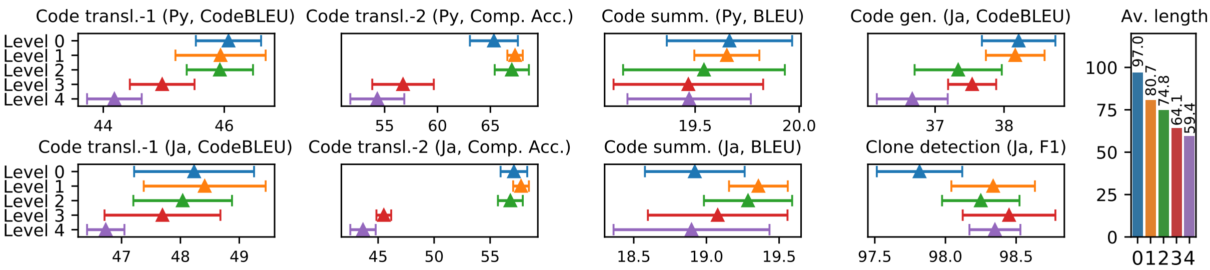


Figure 2: Results on various subtokenization granularity, averaged over 4 finetuning runs (mean ± standard deviation). Level 0 – baseline subtokenization. Numerical data for all plots is given in Appendix.

This reduces the average length by 23% compared to Level 0. The motivation for Level 2 is that a lot of API name tokens almost always go with the dot, e. g. `.join` or `.split` in Python. Figure 2 shows that Level 1 model performs similar or better than Level 0 model in all tasks, and Level 2 performs similar or better than Level 0 in six tasks, marginally worse – in Python code summarization and significantly worse – in Java code generation.

Level 3 makes a step back from Level 4 and restricts the complexity of composite tokens such that each composed token may represent either a simple one-line code pattern or a punctuation combination, but could not combine them. Quantitatively, Level 3 performs generally better than the next Level 4, but (marginally of significantly) worse than the previous Level 2 in six tasks and similar – in two tasks (generation and clone detection).

To sum up, *punctuation combinations (Level 1) results in sequence lengths reduction by 17% without performance drop in all tasks. Length reduction could be increased up to 24% in most tasks by allowing dots attaching to tokens (Level 2) and up to 40% in most code understanding tasks – by allowing arbitrary subtoken combinations (Level 4).* However, one should note that some subtoken combinations are programming language-specific, we investigate the transferability of subtokenizers between programming languages in Section 6.

One of the potential issues with using composite tokens in code-generative tasks is that an inaccurate generation of a “long” token may change all the following generated code. For example, in Java–Python code translation, a cycle which traverses all unique element pairs in an array, converts to

```
for l in range ( 0 , arr_size - 1 ) :
  for r in range ( l + 1 , arr_size ) :
```

While Level 0 model generates exactly the specified cycle and Level 1 model only modifies the first cycle: `range ( arr_size - 1 )`, making it even more concise, Level 3 model generates

```
for l in range ( 0 , arr_size ) :
  for r in range ( 0 , arr_size ) :
```

which results in traversing some elements twice. Here the first cycle was begun with tokens ‘`for l in`’ and ‘`range ( 0 ,`’ and the second cycle was begun with tokens ‘`for r in`’ and `range ( 0 ,` where the latter one repeats the previously used token and starts an incorrect line. However, according to our manual prediction analyses, such an inaccurate generation, if it happens, rarely results in the wrong code and often does not affect code semantics. For example, Level 3 model may generate [`range ( 0 ,`, `n )`] instead of equivalent `range(n)`. Or this model may generate `[ [ 0 ] * column for i in range ( row ) ]` instead of two nested cycles by beginning it with tokens `[ [` and `0 ] *`, resulting in even more concise code.

As for composite tokens in Level 1, they contain only punctuation and are "simpler" than in Level 3. Besides, Level 1 composite tokens more often serve for statement closing, e. g. `)):` at the end of the cycle specification, than for a harder starting of new statements: 46.3% of Level 1 composite tokens contain only closing brackets, 12.8% – only opening brackets and 26.7% contain both. We also check that using punctuation composite tokens does not deteriorate syntactic correctness: in Java-Python code translation-1, Level 0 and Level 1 models generate a similar number of syntactically correct test code snippets: 1226 and 1239 correspondingly. At the same time, for Level 3 model, this quantity only equals 1163.

Berard et al. (2021) point out that in sequence-to-sequence Transformer, the decoder's autoregressive generation is much slower than the encoder's forward pass. Thus we now check that the length statistics of sequences generated by the models comprising composite tokens are close to those of the data. While groundtruth sequences at Levels 1 and 3 are 13.5% and 50% shorter than at Level 0, the generated sequences at these levels are 15% and 40% shorter than sequences generated at Level 0 (numbers for Java-Python translation-1).

## 4 Subtokenization algorithm

Bostrom and Durrett (2020) compare two most popular subtokenization approaches, BPE and UnigramLM (Kudo, 2018), for pretraining of large language models on natural text data. While BPE constructs the vocabulary in the bottom-up fashion, starting from characters and gradually joining them, the UnigramLM algorithm works in the top-down fashion, staring from a large vocabulary and gradually filtering it. The paper finds that UnigramLM outperforms BPE in a range of downstream tasks and suggests several reasons for the superiority of UnigramLM, including better alignment with morphology and the more efficient vocabulary allocation. Since most existing pretrained LMs on source code use BPE (and one model, CuBERT, uses a custom algorithm, see Section 8), we decided to compare two algorithms for source code.

Figure 3 compares BPE and UnigramLM for PLBART. In five tasks, UnigramLM outperforms BPE, with the difference in performance up to one standard deviation, in two tasks UnigramLM performs marginally worse than BPE and in one – significantly worse. Since the average length of two tokenizations is similar, *we recommend using UnigramLM for source code, though the gain in performance is not large*.

Bostrom and Durrett (2020) argue that one of the potential reasons for the superiority of UnigramLM tokenization is that it is better aligned with natural text morphology and thus simplifies the composition of words by parts. We find that a similar effect appears for identifiers in source code: although 80% of identifiers are subtokenized identically by UnigramLM and BPE, for some of the remaining 20%, UnigramLM provides more "reasonable" splits into subtokens, see examples in Table 2. More formally, we observe that UnigramLM subtokenization better resembles splitting into subtokens based on `CamelCase` or `snake_case`, which we call a native subtokenization. To estimate this effect quantitatively, we consider the Python corpus and randomly select a set of 150k identifiers with different UnigramLM and BPE subtokenizations consisting of >= 2 native subtokens, and measure the average Jaccard similarity $J(A, B) = |A \cup B|/|A \cap B|$ between the set of native subtokens and the set of subtokens produced by each subtokenizer. The resulting score for UnigramLM, 26.6%, is much higher than for BPE, 15.2%. As could be observed from the third and the fourth rows in Table 2, sometimes subtokenizers join two native subtokens into one (`isSame`, `GridBag`). If we split each subtoken produced by a tokenizer based on `CamelCase` or `snake_case` to eliminate this effect and then again measure average Jaccard similarities, UnigramLM's score, 55.2%, is still much higher than BPE's, 47.9%, again indicating that UnigramLM's tokenization is better aligned with the native one.

A relatively frequent pattern is that BPE tends to detach the first uppercase letter from

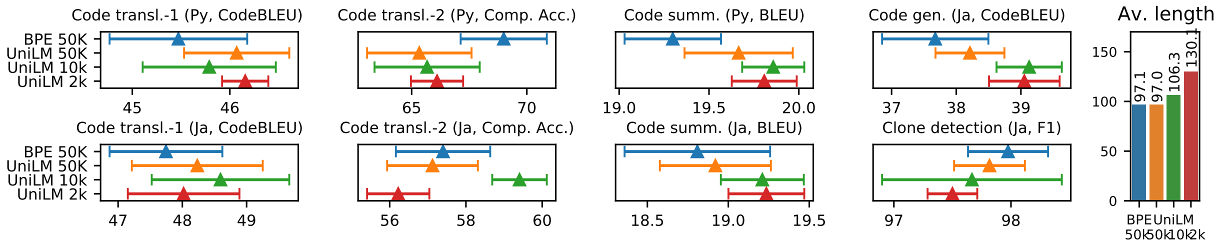


Figure 3: Comparison of BPE and UnigramLM subtokenizers and of several vocabulary sizes. UnigramLM 50K – baseline subtokenization.

| Original token | UnigramLM subtokenization | BPE subtokenization | Native subtokenization (Camel- or snake_case) |
|---|---|---|---|
| fromDottedString | ['from', 'Dotted', 'String'] | ['from', 'Dot', 'ted', 'String'] | ['from', 'Dotted', 'String'] |
| isInstantiated | ['is', 'Instantiate', 'd'] | ['isIn', 'stanti', 'ated'] | ['is', 'Instantiated' ] |
| GridBagConverter | ['Grid', 'Bag', 'Converter'] | ['GridBag', 'Converter'] | ['Grid', 'Bag', 'Converter'] |
| isSameSize Horizontally | ['isSame', 'Size', 'Horizontally'] | ['isSame', 'Size', 'H', 'orizontally'] | ['is', 'Same', 'Size', 'Horizontally'] |
| PA_Hierarchy_ID | ['PA', '_', 'Hierarchy', '_ID'] | ['PA', '_H', 'ierarchy', '_ID'] | ['PA', '_', 'Hierarchy', '_', 'ID'] |

Table 2: Example subtokenization of identifiers by UnigramLM and BPE subtokenizers.

native subtokens (H orizontally in row 4, _H ierarchy in row 5). Among 150k identifiers considered in the previous paragraph, 14.6% of BPE tokenizations contain at least one single uppercase letter X and 4.4% — at least one subtoken of kind _X, while for UnigramLM these scores are significantly less and equal to 11.8% and 1.4% correspondingly. On the other hand, BPE merges two native subtokens more frequently (GridBag in row 3): 45.8% BPE tokenizations contain at least one token which could be split into two or more based on CamelCase, while for UnigramLM this score only equals to 39.2%.

## 5 Vocabulary size

This section studies the effect of vocabulary size, one of the main subtokenizer's hyperparameters, on the downstream quality of PLBART. Though the existing pretrained LMs for code use relatively large vocabularies of 30–50K tokens, we are interested, whether using smaller and less length-efficient vocabularies could result in better performance, and if yes, how large is the length increase.

Figure 3 presents the comparison of PLBARTs trained with vocabulary sizes 50K (large), 10K (medium) and 2K (small). We find that in code translation, all vocabularies lead to similar performance, except Python-Java translation-2 where 10K vocabulary performs best. In code summarization and code generation, small and medium vocabularies outperform the large one by one standard deviation. Finally, in clone detection, increasing the vocabulary size deteriorates quality. At the same time, with the large vocabulary, sequences are shorter than with smaller vocabulary by 9.5% (10K) and 33% (2K). We conclude that *vocabulary size reduction may lead to a slight performance improvement but with sequences elongation*, thus it may be helpful in applications with high cost of errors and weak restrictions on sequences lengths. We note that compared to the BPE 50k subtokenizer which is used in a lot of existing large LMs on source code, the UnigramLM 10k subtokenizer improves performance significantly in three tasks and by one standard deviation – in other three tasks.

Reducing vocabulary size increases the granularity of identifiers subtokenization, e. g. reachable is subtokenized as reachable with 50K vocabulary, reach able – with 10K and re ach able – with 2K. In other words, vocabulary size reduction may be seen as even stronger prohibition of composite tokens than Level 0 in Section 3. Our results on the effectiveness of smaller granularity agree with the machine translation results of (Ding et al., 2019). Programs in code generation and summarization data are more identifier-centered, e. g. the model often needs to choose a correct API based on the natural language description – which seems to be easier by composing from smaller subtokens.

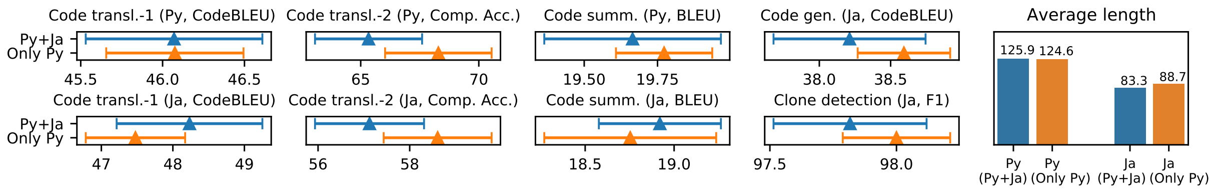


Figure 4: Results of transferability between programming languages. Py+Ja – subtokenizer is trained on all data (baseline), Only Py – subtokenizer is trained on Python and natural language data only.

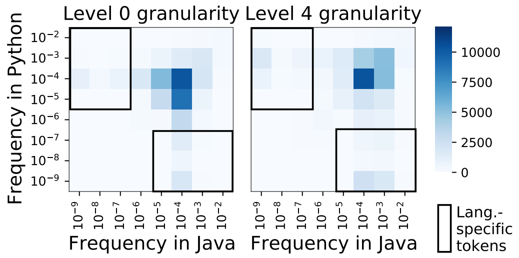


Figure 5: Number of tokens of different frequency in two languages, UnigramLM 50K vocabularies.

On the contrary, in code translation, data is more algorithmic-centered, with mostly short identifiers which are encoded in 1–2 subtokens with all vocabulary sizes. The length increase of 10k vocabulary compared to 50k one is 6–19% in the former two tasks (6% in generation, 19% in summarization) and only 3.5% in the latter one (code-translation-1).

## 6 Transferability between programming languages

Due to the high computational cost of large LM pretraining and relative programming languages similarity, e. g. compared to how dissimilar natural languages could be, pretrained LMs on source code are often used for programming languages that were not considered during pretraining. In this section, we investigate the effect of using a subtokenizer trained on one programming language for another programming language.

Figure 5 visualizes the number of tokens having particular frequencies in Python and Java languages, and black rectangles denote language-specific areas. We find that the baseline Level 0 granularity vocabulary seems to be language-universal: the majority of subtokens have large frequencies in both languages, and only a small number of subtokens, 12.6%, are frequent in one language and rare in another. Interestingly, for Level 4 vocabulary, this quantity is not much larger: 20.1%, though it should include all language-specific composite tokens. As composite tokens occupy almost half of the Level 4 vocabulary, the remaining 30% composite tokens are common for two languages.

Analysing sequence lengths (Figure 4), we observe that training the subtokenizer without Java (*Only Py*) shortens Python sequences marginally and increases Java sequences by 6.5% compared to the baseline subtokenizer trained on all data (*Py+Ja*). The latter happens because some widely used Java identifiers were not merged into single tokens as they are not used in Python; still, the length increase is not so large. For the Level 4 granularity subtokenizer, *Only Py*'s length increase on Java is larger, 13%, since it contains more language-specific composite tokens. However, due to common composite tokens, the resulting Level 4 *Only Py*'s Java average length is still smaller than Level 1 *Only Py*'s Java sequences: 79 vs. 83.

As for downstream performance, using *Only Py* subtokenizer instead of *Py+Ja* changes quality up to one standard deviation and could both increase and decrease it on Java data (quality increase may be caused by the increased subtokenization granularity). Note that we only change subtokenizer configuration – PLBART is still pretrained on all languages, this may happen in practice if LM's developers use the subtokenizer from another project, e. g. for comparison purposes. Summing up, we conclude that *the baseline subtokenizer is universal and, if needed, could be used for other programming languages it was not trained on, with small length increase and slight quality change*.

## 7 Stochastic subtokenization

Kudo (2018), Provilkov et al. (2020) propose stochastic subtokenization to improve the quality of machine translation. For example, BPE-Dropout (Provilkov et al., 2020) skips some subtoken merges during sequence encoding and thus improves the model's capabilities to compose new words. In this section, we investigate the effect of using BPE-Dropout for large LMs pretrained on source code.

Since pretraining a separate LM with BPE-Dropout is computationally expensive in practice, we plug BPE-Dropout into finetuning, for BPE-50k-based PLABRT. We find that *BPE-Dropout improves quality in small-resource Code translation-2 and does not provide consistent improvement in other tasks*. This agrees with results of Wang et al. (2021a) on finetuning BERT with BPE-Dropout on English data and may potentially be improved with their multi-view subword regularization.

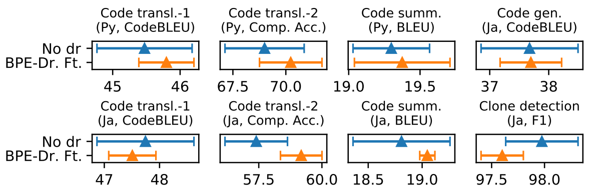


Figure 6: Results of finetuning with BPE dropout.

## 8 Related Work

**Subtokenization studies for NLP.** Subtokenization has become an essential component of modern NLP pipelines and thus — a subject of a line of empirical NLP studies. While word-based models suffer from the out-of-vocabulary problem, subtoken-based (open-vocabulary) as well as char-based approaches cover arbitrary novel words. Among various open-vocabulary approaches, BPE (Sennrich et al., 2016), WordPiece (Wu et al., 2016) and UnigramLM (Bostrom and Durrett, 2020) became most widely used, and UnigramLM was shown to outperform BPE for LM pretraining (Bostrom and Durrett, 2020). A line of studies investigate the optimal granularity of word subtokenization: Ding et al. (2019) find that in Transformer-based neural machine translation, small vocabularies of 0–4K subtokens outperform large ones by up to 4 BLEU, and VOLT (Radford et al., 2018) automates the search of a proper subtoken vocabulary with a proper size by formulating it as an optimal transport problem. The smallest char-based granularity is often avoided because of substantial sequences elongation, but has particular strengths, e. g. much less number of hyperparameters and better robustness, and thus appears to be a promising research direction (Gupta et al., 2019; Clark et al., 2021; Tay et al., 2021). Provilkov et al. (2020); Bostrom and Durrett (2020) propose stochastic subtokenization as a way to improve new words composition and (Wang et al., 2021a) adapt it to pretrained LMs. Finally, an actively studied challenge is that various natural languages need different subtokenization decisions and are hard to subtokenize with one common model (Chung et al., 2020; Rust et al., 2021). Our work investigates most of the specified directions for source code. For a more detailed review on subtokenization, see (Mielke et al., 2021).

**Subtokenization practices in neural source code processing.** Subtokenization was first tested for source code in (Karampatsis et al., 2020) and later used in the majority of Transformer-based models. Almost all LMs pretrained on source code use BPE-like subtokenization with large vocabulary: CodeBERT uses the WordPiece (Wu et al., 2016) algorithm (a modified BPE, 50K), CuBERT – an algorithm from the Tensor2Tensor project (Vaswani et al., 2018) (50K), PLBART and CodeGPT – BPE (50K), CodeT5 – byte-level BPE (32k), DOBF uses a subtokenizetion procedure of either CodeBERT or Roziere et al. (2020) (BPE 64K) for fair comparison. To the best of our knowledge, existing works do not investigate the effect of using composite tokens for source code. Our Level 4 composite tokens are conceptually similar to code idioms used in (Iyer et al., 2019; Shin et al., 2019) for code generation, but the mentioned works develop specific procedures for mining idioms, which need separate implementation, while we rely on the commonly-used subtokenization procedure.

## 9 Conclusion

In this work, we conducted an empirical study of varying subtokenization options for large LMs pretraining on source code. We proposed a punctuation combination approach that shortens sequences by 17% without quality drop and which could be extended with more complex subtoken combinations, shortening lengths up to 40% without performance drop in most code understanding tasks but with significant drop in code-generative tasks. We also showed that using the UnigramLM-10k subtokenizer may be 0.5–2% more effective than the commonly-used BPE 50k, but with 3.5–19% length increase. We call the resulting set of recommendations CodeBPE or CodeUnigramLM. We suggest that future works consider releasing models with both most efficient and most effective subtokenizations. The work's limitation is that we consider only PLBART model, but since other LMs are usually also pretrained using MLM, we assume that our results are transferrable to them as well.

## Broader impact

We do not anticipate any direct negative social impact of our work. However, our results may potentially be used for developing new pretrained LMs for source code, and a detailed discussion on their broader impact is provided in (Chen et al., 2021) (Section 7), e. g. over-reliance on generated code or producing vulnerable code. Unfortunately, our work may cause negative environmental impact because of computation (∼4.3K Tesla A-100 GPU hours and ∼4K Tesla V-100 GPU hours).

## References


Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021a. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online. Association for Computational Linguistics.

Wasi Uddin Ahmad, Md Golam Rahman Tushar, Saikat Chakraborty, and Kai-Wei Chang. 2021b. Avatar: A parallel corpus for java-python program translation.

Alexandre Berard, Dain Lee, Stephane Clinchant, Kweonwoo Jung, and Vassilina Nikoulina. 2021. Efficient inference for multilingual neural machine translation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8563–8583, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Hyung Won Chung, Dan Garrette, Kiat Chuan Tan, and Jason Riesa. 2020. Improving multilingual models with language-clustered vocabularies. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 4536–4546, Online. Association for Computational Linguistics.

Jonathan H. Clark, Dan Garrette, Iulia Turc, and John Wieting. 2021. Canine: Pre-training an efficient tokenization-free encoder for language representation.

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.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie LIU, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcode{bert}: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Rohit Gupta, Laurent Besacier, Marc Dymetman, and Matthias Gallé. 2019. Character-based nmt with transformer. *ArXiv*, abs/1911.04997.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. Codesearchnet challenge: Evaluating the state of semantic code search.

Srinivasan Iyer, Alvin Cheung, and Luke Zettlemoyer. 2019. Learning programmatic idioms for scalable semantic parsing. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 5426–5435, Hong Kong, China. Association for Computational Linguistics.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652, Brussels, Belgium. Association for Computational Linguistics.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *Proceedings of* *the 37th International Conference on Machine Learning, ICML 2020, 12-18 July 2020*, Proceedings of Machine Learning Research. PMLR.

Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code!= big vocabulary: Open-vocabulary models for source code. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1073–1085, New York, NY, USA. Association for Computing Machinery.

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.

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

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7871–7880, Online. Association for Computational Linguistics.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *ArXiv*, abs/1907.11692.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664.

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

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

Alec Radford and Karthik Narasimhan. 2018. Improving language understanding by generative pretraining.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2018. Language models are unsupervised multitask learners.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, M. Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *ArXiv*, abs/2009.10297.

Baptiste Roziere, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. In *Advances in Neural Information Processing Systems*, volume 33, pages 20601–20611. Curran Associates, Inc.

Baptiste Roziere, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. 2021. Dobf: A deobfuscation pre-training objective for programming languages. In *Advances in Neural Information Processing Systems (NeurIPS 2021)*, Online.

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

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.

Richard Shin, Miltiadis Allamanis, Marc Brockschmidt, and Oleksandr Polozov. 2019. *Program Synthesis and Semantic Parsing with Learned Code Idioms*. Curran Associates Inc., Red Hook, NY, USA.

Jeffrey Svajlenko and Chanchal K. Roy. 2015. Evaluating clone detection tools with bigclonebench. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 131–140.

Yi Tay, Vinh Q. Tran, Sebastian Ruder, Jai Gupta, Hyung Won Chung, Dara Bahri, Zhen Qin, Simon Baumgartner, Cong Yu, and Donald Metzler. 2021. Charformer: Fast character transformers via gradient-based subword tokenization.

Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan Gomez, Stephan Gouws, Llion Jones, Łukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, and Jakob Uszkoreit. 2018. Tensor2Tensor for neural machine translation. In *Proceedings of the 13th Conference of the Association for Machine Translation in the Americas (Volume 1: Research Track)*, pages 193–199, Boston, MA. Association for Machine Translation in the Americas.

Xinyi Wang, Sebastian Ruder, and Graham Neubig. 2021a. Multi-view subword regularization. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 473–482, Online. Association for Computational Linguistics.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021b. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144.

## A Appendix

Table 3 presents the numerical results for figures in the main text.

| **Subtokenizer** | **CT1 (Py)** | **CT1 (Ja)** | **CT2 (Py)** | **CT2 (Ja)** | **CS (Py)** | **CS (Ja)** | **CG (Ja)** | **CD (Ja)** |
|---|---|---|---|---|---|---|---|---|
| UnigramLM 50k Level 0 | 46.1 | 48.2 | 65.3 | 57.1 | 19.7 | 18.9 | 38.2 | 97.8 |
| UnigramLM 50k Level 1 | 45.9 | 48.4 | 67.3 | 57.8 | 19.7 | 19.4 | 38.2 | 98.3 |
| UnigramLM 50k Level 2 | 45.9 | 48.0 | 67.0 | 56.8 | 19.5 | 19.3 | 37.3 | 98.2 |
| UnigramLM 50k Level 3 | 45.0 | 47.7 | 56.7 | 45.5 | 19.5 | 19.1 | 37.5 | 98.5 |
| UnigramLM 50k Level 4 | 44.2 | 46.7 | 54.3 | 43.7 | 19.5 | 18.9 | 36.7 | 98.3 |
| BPE 50K Level 0 | 45.5 | 47.7 | 69.0 | 57.4 | 19.3 | 18.8 | 37.7 | 98.0 |
| UnigramLM 10k Level 0 | 45.8 | 48.6 | 65.7 | 59.4 | 19.9 | 19.2 | 39.1 | 97.7 |
| UnigramLM 2k Level 0 | 46.2 | 48.0 | 66.1 | 56.2 | 19.8 | 19.2 | 39.1 | 97.5 |
| UnigramLM 50k Level 0 (Only Py) | 46.1 | 47.5 | 68.3 | 58.6 | 19.8 | 18.8 | 38.6 | 98.0 |
| BPE 50K Level 0 + BPE-Dropout | 45.8 | 47.5 | 70.2 | 59.2 | 19.4 | 19.0 | 37.7 | 97.6 |

Table 3: Numerical data for figures in the main text. CT1: Code Translation-1 (CodeBLEU), CT2: Code Translation 2 (Computational Accuracy), CS: Code Summarization (BLEU), CG: Code Generation (CodeBLEU), CD: Clone Detection (F1). Py – Python, Ja – Java.