Big Code != Big Vocabulary: 
Open-Vocabulary Models for Source Code

Rafael-Michael Karampatsis  
University of Edinburgh  
Edinburgh, United Kingdom

Hlib Babii  
Free University of Bozen-Bolzano  
Bozen-Bolzano, Italy

Romain Robbes  
Free University of Bozen-Bolzano  
Bozen-Bolzano, Italy

Charles Sutton  
Google Research and University of Edinburgh  
Mountain View, CA, United States

Andrea Janes  
Free University of Bozen-Bolzano  
Bozen-Bolzano, Italy

1 INTRODUCTION

Many works have taken advantage of the “naturalness” of software [44] to assist software engineering tasks, including code completion [76], improving code readability [2], program repair [20, 78], identifying buggy code [75] and API migration [38], among many others [4]. These approaches analyze large amounts of source code, ranging from hundreds to thousands of software projects, building machine learning models of source code properties, inspired by techniques from natural language processing (NLP).

When applying any NLP method to create any type of software development tool, a crucial early decision is how to model software’s vocabulary. This is all the more important because, unlike in natural language, software developers are free to create any identifiers they like, and can make them arbitrarily complex. Because of this fundamental fact, any model that is trained on a large-scale software corpus has to deal with an extremely large and sparse vocabulary (Section 2). Rare words can not be modelled effectively. Furthermore, if identifiers were not observed in the training set, many classes of models cannot predict them, which is known as the out-of-vocabulary (OOV) problem. Hellendoorn and Devanbu observe this issue for the task of language modeling, showing that a neural language model (NLM) has difficulties scaling beyond as few as a hundred projects [41]. Given that neural approaches are the state-of-the-art in NLP, finding ways to scale them to a larger software corpus is a very important goal.

Our first contribution is a thorough study of the effects of the vocabulary design choices that must be made when creating any NLP model of software (Section 4). The vocabulary design choices we study include how to handle comments, string literals, and whitespace; whether to filter out infrequent tokens; and whether and how to split compound tokens, such as names that contain camel case and underscores. We examine how these choices affect the vocabulary size, which affects the scalability of models, and how they affect the OOV rate, that is, how often the vocabulary fails to include names that appear in new projects. We find that the choices have a large impact, leading to variations in vocabulary size of up to three orders of magnitude. However, we find that the most common ways to reduce vocabulary that were previously considered in the software engineering literature, such as splitting identifiers according to underscores and case, are not enough to obtain a vocabulary of a manageable size; advanced approaches such as adaptations of the Byte-Pair Encoding (BPE) algorithm [34, 79] are needed to reach this goal and deal with the OOV problem.
This empirical study motivates our second contribution. Drawing on our results, we develop a large-scale open-vocabulary NLM for source code (Section 5). To our knowledge, this is the first BPE NLM for source code reported in the literature. This NLM model leverages BPE, beam search, and caching to both keep vocabulary size low and successfully predict OOV tokens. We show that this NLM is able to scale: we train it on up to 13,362 software projects, yielding the largest NLM trained on source code we are aware of.

Finally, in our third contribution we extensively evaluate our NLM (Sections 6–8). We show that the open-vocabulary NLM outperforms both n-gram LMs and closed vocabulary NLMs for the task of code completion for several languages (Java, C, and Python). To show that improvement in language modelling transfers to downstream SE tasks, we conduct an experiment similar to Ray et al. [75], who showed that language models can be used to highlight buggy code. Indeed, we find that our open-vocabulary NLM is more effective than previous LMs at highlighting buggy code.

More broadly, these contributions may impact future development software tools. First, source code LMs have been used in a diverse variety of tools well beyond the obvious application of autocompletion, ranging from code readability [2] to program repair [20]. Our improved NLM could lead to improvements to all of these tools. Second, recent results in NLP [28, 46, 69] show that NLMs can be used as upstream tasks in transfer learning, leading to state-of-the-art improvement in downstream tasks: for instance, a model can be pre-trained as an NLM, and later on fine-tuned as a classifier. Improved NLM architectures could lead to improved downstream classifiers, especially if the labelled data is scarce. While transfer learning from language models has been applied in software engineering [77], it has not been applied to source code due to the aforementioned vocabulary issues. Finally, the general insights about vocabulary design that we study are not specific to NLMs, but arise whenever we build development tools by applying NLP methods to source code.

We conclude the paper in Section 9, and briefly describe the artifacts used in this work and how to obtain them in Section 10.

## 2 BACKGROUND AND RELATED WORK

We first note that this work is a consolidation of two unpublished works originally conducted independently: one work focused on the tokens in the sequence. These had success in NLP applications, but have two issues. First, they operate on small amounts of previous context, with \( n \) often ranging from 3 to 6 (e.g. \( n = 6 \) for Java [41]). Increasing \( n \) does not scale well if the vocabulary is large: for a vocabulary of size \( m \), there are \( m^n \) possible \( n \)-grams. Second, they suffer from data sparsity: not all possible \( n \)-grams exist in the corpus. Smoothing [19] alleviates—but does not eliminate—the issue.

The current state-of-the-art in NLP is neural language models (NLM) [12]. NLMs represent words in a continuous vector space, such that words that are semantically similar are close in vector space [64], allowing the model to infer relationships between words, even if they do not appear in a specific context during training. This allows these models to better deal with data sparsity, leading to better performance. Current NLMs are based on architectures such as recurrent neural networks (RNN) [63], long short-term memory (LSTM) [45], or Transformer [85] that model long range dependencies: a study of LSTM NLMs showed that they use context as large as 250 words [56], much longer than \( n \)-grams.

### 2.2 Difficulties with Large Vocabularies

ML models in general, and NLMs in particular, do not handle large vocabularies well. This is for several reasons:

**Scalability.** During pre-processing, each word is converted to a numerical representation, first via one-hot-encoding, producing (sparse) vectors of length equal to the vocabulary. NLMs then convert these to word embeddings, dense word vectors of much smaller dimensions (usually in the hundreds), in their first layer. For a vocabulary of size \( m \) and embeddings of size \( n \), the embedding layer is a dense matrix of size \( m \times n \). A large \( m \) (e.g., 100,000 or more) affects the memory required by the model as well as the amount of computation required for training. The output of an NLM is a prediction over the next token, which is a probability distribution over the entire vocabulary. This must be computed once for each token in the training corpus many times during training. This can be prohibitively slow for large vocabularies [16, 52].

**Out-of-vocabulary (OOV).** In traditional, closed-vocabulary models, the vocabulary must be known in advance and will be built based on the training corpus. Any new word encountered at test time, called out-of-vocabulary words, will not be able to be one-hot encoded as the resulting vector would exceed the expected dimensions. A common workaround is to have a specific unknown token, and replace any word not previously seen by this token. This loses information, making the NLM unable to predict any new token, which is particularly problematic for source code.

**Rare Words.** Deriving meaningful embeddings for rare words is difficult since there is very little data to work with. Gong et al. show that the property that semantically similar words have similar embeddings does not hold for rare words: they hypothesize that since the words are rarely seen, the embeddings are rarely updated and thus stay close to their initialized values [35]. This issue is likely to impact performance: a very large vocabulary has been shown to negatively impact it, particularly with OOV words [51].

### 2.3 Handling Large Vocabularies in NLP

An open vocabulary model is not restricted to a fixed-sized vocabulary determined at training time. For instance, a character LM
predicts each word letter by letter: its vocabulary is the set of characters; the OOV issue vanishes. However, it needs to model longer dependencies than a word NLM, impacting performance. Models using subword units, or subwords, combine the strengths of character and token LMs. A subword unit is a sequence of characters that occurs as a subsequence of some token in the training set; the model outputs a sequence of subword units instead of a sequence of tokens. Many NLP models have used linguistically-motivated subwords [11, 24, 59, 65]. Mikolov et al. found that subword models improved on character models [65]. Sennrich et al. adapt the Byte-Pair Encoding (BPE) algorithm to decompose words in subwords, improving rare word translation [79]. Kim et al. combine a character CNN with a NLM [57]. Vania and Lopez compare LMs (words, morphs, character n-grams, BPE) on several languages [83].

Another approach to the OOV problem are cache models and copy mechanisms [5, 36, 62], which allow the model to re-use words that have appeared previously. This helps with the OOV problem, because such models can copy words that are not in their fixed vocabulary, but it does not help the first time an OOV word appears.

2.4 Language Modeling and Vocabulary in SE

Language Models in Software Engineering (SE). Seminal studies have laid the groundwork for the use of language models on source code: Gabel and Su show that software is very repetitive [33], which motivates the use of statistical modelling for code. Hindle et al. [44] build language models of source code, finding applications in code completion. Nguyen et al. [68] augmented n-gram LMs with semantic information such as the role of a token in the program, e.g., variable, operator, etc. Tu et al. [81] find that software is even more repetitive taking local context into account. Rahman et al. find that while some aspects of software are not as repetitive as previously thought (non-syntactic elements), others are even more so (API sequences) [74]. Other models of source code include probabilistic higher order grammars (PHOG) [14], which use ASTs, and several types of RNNs, including LSTMs [26, 41, 88].

SE Applications of Language Models. Probabilistic code models have enabled many applications in software engineering (see Alamanis et al. [4] for a survey). One example is recommender systems aiming to aid developers in writing or maintaining code: Hindle et al. used a token-level LM for code completion [44], while later, Franks et al. improved on performance with Tu’s cache [81] and built a code suggestion tool for Eclipse [32]. Another application are recommendation systems for variable, method, and class names [2, 3, 5] that employ relevant code tokens as the LM context. Campbell et al. [18] used n-gram language models to detect syntax error locations in Java code, and later used an NLM for the same purpose [78]. Ray et al. [75] showed that buggy code has on average lower probability than correct code, and that LMs can spot defects as effectively as popular tools such as FindBugs.

Several approaches use neural machine translation, in which an encoder LM is paired to a decoder LM. Examples include recovering names from minified Javascript code [10, 84], or from decompiled C code [50]. Other applications include program repair [20], learning code changes [82], or generating source code comments [47]. Gu et al. [37] generate API usage sequences for a given natural language query. They then learn joint semantic representations of bilingual API call sequences to support API call migration [38]. Yin et al. [90] mine pairs of natural language and code from Stack Overflow to support tasks such as code synthesis from natural language.

Large vocabularies in SE. The majority of models of source code used closed vocabulary models. Hellendoorn and Devanbu rightly notice that LMs trained on a software corpus would struggle due to vocabulary size [41], because identifiers, which are the bulk of source code, can be arbitrarily complex, and are often compound words (e.g., thisIdentifierHas6WordsAnd2Numbers), causing an explosion of possible identifiers. To produce an NLM that can be trained in a reasonable amount of time, Hellendoorn and Devanbu impose drastic restrictions which would be expected to reduce predictive accuracy, restricting the training set to 1% of the original corpus [6] and the vocabulary to only include words which occur more than 5 times. Even so, the resulting vocabulary size is still exceeds 76,000 words. Similarly, Pradel and Sen [70] had a large vocabulary of 2.4 million unique tokens: they limited it to the 10,000 most common tokens to reduce inaccuracies due to rare words.

To limit this issue, previous work has segmented identifiers via a heuristic called convention splitting, which splits identifiers on camel case and underscores [3]. Even though this segmentation can handle some OOV tokens, it is limited to combinations of subtokens appearing in the training set and thus unable to achieve a truly open vocabulary. Additionally, many of these subtokens are still infrequent, which hinders the model’s ability to assign high scores to their compositions. For example, despite using convention splitting, the implementation of code2seq from Alon et al. [8] only keeps the 190,000 most common vocabulary words.

Several studies have empirically compared different techniques for automatically splitting identifiers [30, 43]. These works consider the somewhat different problem of splitting identifiers into words in a way that matches human judgment. Subword units may not necessarily produce words that humans recognize, but they can be trivially reassembled into complete tokens before they are shown to a developer. Stemming [89] has also been used to reduce the number of vocabulary words by only keeping their roots; this is however destructive. Malik et al. combined convention splitting and stemming for type prediction [60].

Few SE approaches use caches. Tu et al. [81] and Hellendoorn and Devanbu [41] use n-gram caches. Li et al. augment an RNN with a copy mechanism based on pointer networks [86] to improve OOV code completion [58]. While it can reuse an OOV word after seeing it, it cannot predict the word’s first use, learn its representation, or learn its dependencies, unlike our model. Copy mechanisms were also used for program repair [20], and method naming [5].

3 DATASETS

We use code corpora from three popular programming languages: Java, C, and Python. We choose these languages because they have differences that could affect the performance of LMs. Java has extensively been used in related work [6, 26, 41, 44, 68, 81]. Unlike Java, C is procedural, and makes it possible to write very terse code.1 Python is a multi-paradigm dynamic language with little use of static typing. For Java we used the Java Github corpus of Alamanis et al. [6], also used in [41]. The C and Python corpora

---

1For examples, see https://www.ioccc.org/.
were mined following the procedure described in [6]; the C corpus was mined in [29] and the Python corpus was mined in [31]. For lexical analysis we used the Java lexer implemented in [41]²; for C and Python we used the Pygments³ library. Descriptive statistics are in Table 1.

For Python and C we sampled 1% of the corpus for validation and 1% for testing. Another 10% of the corpus was sampled as a separate data set to learn subword encodings with BPE. The rest of the data was used for training. We also report results on a smaller subset of 2% of our full training set. For Java, we used a slightly different procedure to make our experiment comparable to a previous study [41]. We divide the data into five subsets as in the other two languages. The validation and test sets are the same as in [41], and our "small train" set is the same as their training set. To obtain the full Java train set, we collect all of the files in the Java Github corpus that do not occur in the validation or test set. Of these, we sampled 1000 random projects for the subword encoding data set, and the remaining projects were used as the full train set.

In the vocabulary study, both training sets and test sets are used. To train LMs, we preprocess the corpora to match [41], replacing non-ASCII character sequences such as Chinese ideograms inside strings with a special token (<non-en>), removing comments, and keeping strings. Note that the lexer in [41] replaces all strings with <string>, or "Heuristiken" are misclassified. Non-English words are replaced with a placeholder token like <non-en>. Even then, the vocabulary shrinks by 35%, 76%, and 83%.

1.1 Filter vocabulary

We study a series of modeling choices for source code vocabulary. These choices may be implicitly made by researchers, with or without evaluating alternatives; they may not always be documented in their studies. By making the choices explicit, we can study their impact on the vocabulary. We report results on Java; similar results can be observed for C and Python. Our evaluation criteria are:

Scalability Training of models should scale to thousands of projects. Scalability is influenced by the vocabulary size (number of unique words) and the corpus size (number of tokens). We report both metrics on our full Java training set, and compare them to a baseline with percentages. For instance: 11.6M, 100% and 2.43B, 100%.

Information loss Models should be able to represent the original input as much as possible; out-of-vocabulary (OOV) tokens are particularly undesirable. We build a vocabulary on the training set, and compare it with the test set vocabulary; we report the percentage of new vocabulary words seen in the test set. As large vocabularies do not scale, we report OOV for the unfiltered vocabulary, and a smaller vocabulary (the 75,000 most frequent words, as in [41]). To show trends, we also plot OOV for: full vocabulary, 200K, 100K, 75K, 50K, and 25K. Such as: 42%, 79%, 83%.

Word frequency As rare words have worse representations than frequent ones [35], increasing word frequency is desirable. Different modelling choices can increase or decrease the number of rare words. We report the percentage of the vocabulary that has a frequency of 10 or less, and plot a bar chart showing the percentage of vocabulary with frequencies of 1000+, 1000–101, 101–11, 10–2, and 1. For instance: 83%, 79%.

Baseline: 11.6M, 100% • 2.43B, 100% • 42%, 79%, 83%. Our baseline is a vocabulary of unsplit tokens, except strings and comments that are split by whitespace (not doing so roughly doubles the vocabulary). This vocabulary is extremely large: more than 11 million unique words on Java-large. The OOV rate on the test set exceeds 40% with the full vocabulary, showing that developers do create many new identifiers. The most common way to shrink vocabulary is to replace infrequent tokens with <unk>. Doing so further worsens OOV issues: after reducing the vocabulary to a more manageable 75K, close to 80% of the test vocabulary is unseen in the training set. Many words are infrequent: 83% of vocabulary words have a frequency of 10 or less, with 25% occurring only once.

4.1 Filtering the vocabulary

Simplest is to filter vocabulary items that are deemed less important. Filtering is destructive: it thus needs to be thoroughly justified.

- English. 11.4M, 98% • 2.43B, 100% • 35%, 76%, 83%. Source code can contain many non-English words in identifiers, strings, and comments, either because developers use other languages, or for testing or internationalization purposes. Handling multilingual corpora is an NLP research topic in itself; we evaluate the simplifying assumption to limit a corpus to English. This is not trivial: dictionary-based heuristics have too many false positives (e.g. acronyms). We use a simple heuristic: a word is non-English if it contains non-ASCII characters. This is imperfect: "café", "naïve", or "Heuristiken" are missclassified. Non-English words are replaced with a <non-en> placeholder. Even then, the vocabulary shrinks by only 2%, while OOV drops by only 3% at 75K.

- Whitespace. 11.4M, 98% • 1.89B, 78% • 35%, 76%, 83%. Some applications (e.g., pretty-printers [3]) may care about the layout of source code. Others may not, giving importance only syntactic or semantic aspects (unless code layout is syntactically important, such as in Python). Filtering out whitespace reduces the vocabulary only by a handful of tokens, but reduces corpus size by 22% (1.89B tokens).

- Comments 10.8M, 93% • 1.26B, 52% • 38%, 78%, 83%. Comments often contain natural language, which is much less repetitive than code. While tasks such as detecting self-admitted technical debt [25] rely on comments, others do not. Replacing comments by placeholder tokens (e.g., <comment>) significantly reduces corpus size (a further 26%), but its effect on vocabulary is limited (6%, given that comments are already split on whitespace).

- Strings. 9.5M, 82% • 1.15B, 47% • 39%, 78%, 83%. String literals can be filtered, replacing them by a placeholder token like <string>. This does not reduce corpus size as

<table>
<thead>
<tr>
<th>Table 1: Corpus statistics for each code corpus.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
</tr>
<tr>
<td>Tokens</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>Full</td>
</tr>
<tr>
<td>Small</td>
</tr>
<tr>
<td>BPE</td>
</tr>
<tr>
<td>Valid.</td>
</tr>
<tr>
<td>Test</td>
</tr>
</tbody>
</table>
much (a further 5%), but shrinks vocabulary a further 11%, close to 9.5 million words. This is still extremely large. We also evaluate the configuration used in [41]: strings are kept, unsplit, but strings longer than 15 characters are replaced by the empty string. For consistency with previous work, we use it as new baseline. It increases vocabulary, OOV and infrequent tokens rate: 10.9M, 94% • 1.15B, 47% • 39%, 88% • 84%. Full token vocabularies range in the millions, and hence do not scale. OOV and frequency issues are extremely important.

4.2 Word Splitting

Identifiers are the bulk of source code and its vocabulary. While new identifiers can be created at will, developers tend to follow conventions. When an identifier is made of several words, in most conventions, the words are visually separated to ease reading, either in camelCase or in snake_case [15]. Thus, an effective way to reduce vocabulary is to split compound words according to these word delimiters, as was done by Allamanis et al. [3].

The decision whether to split compound words or not has important ramifications. First, it introduces additional complexity: the LM can no longer rely on the assumption that source code is a sequence of tokens. Instead, compound words are predicted as a sequence of subtokens, albeit in a smaller vocabulary. Second, subtokens increase the length of the sequences, making it harder to relate the current subtokens to the past context, as it increases in size. This makes the approach unviable for n-grams as n would need to increase significantly to compensate.

Splitting tokens has advantages: most obviously, the vocabulary can be much smaller. Consequently, the OOV rate is reduced. Third, a model may infer relationships between subtokens, even if the composed word is rare, as the subtokens are more common than the composed word. Finally, using subtokens allows a model to suggest neologisms, tokens unseen in the training data [3].

Splitting. 1.27M, 12% • 1.81B, 157% • 8%, 20%, 81% • 81%, 81% • 81%, 81% • 81%. Word splitting via conventions drastically reduces the vocabulary, by a close to an order of magnitude (slightly more than a million words), at the cost of increasing corpus size by 57%. The impact on the OOV rate is also very large, as it decreases by a factor of 5 (in the unfiltered case; for a vocabulary of 75K it is a factor of 4). However, the effect on word frequency is limited, with only 3% more words that are more frequent than 10 occurrences.

Case. 1.09M, 10% • 2.16B, 187% • 9%, 21%, 83% • 83%. Most commonly, words in different case (e.g. value, Value, VALUE) will be distinct words for the LM. This could increase the vocabulary, but removing case loses information. A possible solution is to encode case information in separator tokens (e.g., Value becomes <Upper> value; VALUE becomes <UPPER> value), at the cost of increasing sequence length. Case-insensitivity does decrease the vocabulary, but not by much (a further 2%), while corpus size increases significantly (a further 30%). Thus, our following configurations do not adopt it: our new baseline keeps case.

Word splitting is effective, but the vocabulary is still large (a million words). OOV and frequency issues are still important.

4.3 Subword splitting

As splitting on conventions is not enough, we explore further.

Numbers. 795K, 63% • 1.85B, 102% • 6%, 18%, 72% • 72%, 72%. Numeric literals are responsible for a large proportion of the vocabulary, yet their vocabulary is very limited. Thus, an alternative to filtering them out is to model them as a sequence of digits and characters. This yields a considerable decrease in vocabulary with our previous baseline (37%), for only 2% increase in corpus size. For OOV, there is a slight improvement for a 75K vocabulary (2%), as well as for frequency (28% of words occur 10 times or more).

Spiral. 476K, 37% • 1.89B, 104% • 3%, 9%, 70% • 70%. Several approaches exist that split a token into subtokens, but go beyond conventions by using Mining Software Repositories techniques, such as Samurai [30], LINSSEN [23], Spiral [48], or even neural approaches [61]. We applied the Spiral token splitter, which is the state of the art. We observed a further 26% reduction of the vocabulary, for a 2% increase in corpus size compared to number splitting. Spiral was also very effective in terms of OOV, with 9% of unseen word when the vocabulary is limited to 75K, and 3% when unfiltered (476K words). The impact on frequency was limited. Even if this is encouraging, the OOV rate is still high.

Other approaches. Stemming [89] can reduce vocabulary size, but loses information: it is not always obvious how to recover the original word from its stem. We found that applying stemming can further reduce vocabulary by 5%, which does not appear to be a worthwhile tradeoff given the loss of information. Another option is character models that achieve an open vocabulary by predicting the source file one character at a time. OOV issues vanish, but unfortunately, this drastically inflates sequence lengths, so a character model is not desirable.

While these strategies are effective, they do not go far enough; vocabulary stays in the hundreds of thousands range. There are still OOV issues for unseen data; most words are uncommon.

4.4 Subword splitting with BPE

The final alternative we evaluate is subword segmentation via Byte-Pair Encoding (BPE). BPE is an algorithm originally designed for data compression, in which bytes that are not used in the data replace the most frequently occurring byte pairs or sequences [34]. In subword segmentation, this corpus is represented as a sequence of subwords. Special end-of-token <t> symbols are added to allow us to convert from a sequence of subword units back into a sequence of tokens with ease. The approach was adapted to build NMT vocabularies [79]: the most frequently occurring sequences of characters are merged to form new vocabulary words.

BPE builds up the vocabulary of subwords iteratively, at each iteration a training corpus is segmented according to the current vocabulary. The initial vocabulary contains all characters in the data set and <t>, and the corpus is split into characters and <t>. Then, all symbol pairs appearing in the vocabulary are counted. All the appearances of the most frequent pair (S1, S2) are replaced with a unique new single symbol S1S2, which is added to the vocabulary, without removing S1 or S2 (which may still appear alone). This procedure is called a merge operation. The algorithm stops after a given maximum number n of merge operations; this is the only
words that are in the hundreds, and 1% less than ten. Lower amounts is shown in Figure 1. We computed BPE for 1K, 2K, 5K, 10K examples include rare words due to typos (in producing longer sequences of shorter units as expected. Some exceptions include rare words (split and suffix (Grid • ify</t>), but some acronyms may be unexpectedly split (IB • AN</t>)). Further, BPE can split words correctly without case information (ht http • client • lib</t>, at 5K).

BPE shrinks source code vocabulary very effectively. Moreover, most of the vocabulary is frequent, improving embeddings.

5 NEURAL LANGUAGE MODEL FOR CODE

We present our NLM for code based on subword units, which is based on a Recurrent Neural Network (RNN). RNN LMs scan an input sequence forward one token at a time, predicting a distribution over each token given all of the previous ones. RNNs with gated units can learn when to forget information from the hidden state and take newer, more important information into account [45]. Among various gated units, GRUs [21] have been shown to perform comparably to LSTMs [45] in different applications [22].

We intentionally selected a small model as our base model: a single layer GRU NLM built upon subword units learned from BPE (Section 4.4). For each vocabulary entry we learn a continuous representation of 512 features, while the GRU state is of the same size. In all our experiments we used a learning rate of 0.1, dropout of 0.5 [80] and a maximum of 50 epochs of stochastic gradient descent stops. During training of the global model we unroll the GRU for 200 timesteps, following [56]. Our implementation is open source (GitHub URL omitted for review). We also experiment with larger capacity models (2048 hidden features and GRU state).

5.1 Selecting Subword Units with BPE

In our code LM, we address vocabulary issues by having the model predict subwords rather than full tokens at each time step. Subwords are inferred by BPE (Section 4.4) on a held out dataset of projects that are separate from the training, validation, and test sets. We experimented with three encoding sizes, i.e., the maximum number of merge operations: 2000, 5000, and 10000. To train the LM, we first segment the train, validation, and test sets using the learned encoding. We transform each token into a character sequence, adding </t> after every token. Then we apply in order the merge operations from BPE to merge the characters into subword units in the vocabulary. As in [79] we do not merge pairs that case information (</t> or foreign words (v • orm • er • k • medi • en • au • f • lis • ter</t>)). Some rare words are split in root and suffix (split (IB • AN</t>)). Further, BPE can split words correctly without case information (ht http • client • lib</t>, at 5K).

BPE shrinks source code vocabulary very effectively. Moreover, most of the vocabulary is frequent, improving embeddings.

5.2 Predicting Tokens from Subword Units

Autocompletion algorithms present a ranked list of k predicted tokens rather than a single best prediction. With a model based on subword units, it is not obvious how to generate the top k predictions, because a single token could be made from many subword

Figure 1: Example of Java code as a list of subword units.

parameter. The final output of the algorithm is (1) the new vocabulary, which contains all the initial characters plus the symbols created from the merge operations, and (2) the ordered list of merge operations performed in each iteration. New data is segmented by splitting it into characters and merging in the same order.

BPE has several advantages. First, no word is OOV; the vocabulary always contains all single characters, so unknown words at test time can be represented using those subwords, if no longer subwords apply. Second, the vocabulary dynamically adapts to the frequency of the sequences: common sequences will be represented by a single word (eg. exception), while rare ones will be segmented into more common subword units (such as roots, prefixes and suffixes); this helps with sparsity issues. Finally, BPE allows for fine-grained control of vocabulary size, by tuning the number of merge operations. A larger vocabulary will have more complete words and less sequences, smaller ones will have longer sequences. An example of a Java code snippet segmented into subwords is shown in Figure 1. We computed BPE for 1K, 2K, 5K, 10K and 20K merges, on a held-out set of 1K project.

BPE Subwords. 10K, 1% • 1.57B, 137% • 0%, 0%, 1%, 1%

We apply BPE (10K merges) to our Java corpus with preprocessing as in [41], which we use as a baseline for comparison. As expected, the OOV issues vanish, even for an extremely small vocabulary. The corpus size grows, but not more than previous choices we explored. Since BPE merges based on frequency, the resulting subtokens, no matter their size, are frequent: more than 97% of the remaining words occur more than 1,000 times in the corpus, with very few words that are in the hundreds, and 1% less than ten. Lower amounts of merges result in a smaller vocabulary, at the cost of a larger corpus size. Our largest BPE vocabulary, 20K, is 575 times smaller than our initial baseline; our smallest is 11,500 times smaller.4

Qualitative examination. While the goal of BPE is not to produce human-readable tokens, we examine how closely the splits BPE produces match human ones. We inspected 110 random identifiers, and provide anecdotal evidence of the types of splits produced by BPE. Our goal is not to provide strong evidence, but rather to give a sense to the reader of what BPE splits look like in practice.

While some subwords are readable at BPE 1K (File • Output • Service</t>), some subwords are not (Default • M • ut • able • Tre • e • Node</t>), but look good at 5K (Default • Mutable • TreeNode</t>). BPE handles rare words gracefully, producing longer sequences of shorter units as expected. Some examples include rare words due to typos (in • cul • ded •

4Note that including non-ASCII characters grows the vocabulary by ≈ 5,000 words in each case, a solution is to apply BPE at the byte level, as done in [71]

5.5 Predicting Tokens from Subword Units

Autocompletion algorithms present a ranked list of k predicted tokens rather than a single best prediction. With a model based on subword units, it is not obvious how to generate the top k predictions, because a single token could be made from many subword
units. We approximate these using a custom variation of the beam search algorithm. If the beam is large enough the algorithm can give a good approximation of the top-\(k\) complete tokens.

The NLM defines a probability \(p(s_1 \ldots s_N)\) for any subword unit sequence. The goal of the beam search is: given a history \(s_1 \ldots s_N\) of subwords that already appear in a source file, predict the next most likely complete token. A complete token is a sequence of subword units \(s_1 \ldots s_M\) that comprise exactly one token: that is, \(s_M\) ends with \(</t>\) and none of the earlier subword units do. Beam search finds the \(k\) highest probability complete tokens, where we denote a single token as the sequence of units \(s_1 \ldots s_M\), that maximize the model’s probability \(p(w_1 \ldots w_M|s_1 \ldots s_N)\). Importantly, the length \(M\) of the new complete token is not fixed in advance, but the goal is to search over complete tokens of different length.

Given a value of \(k\) and a beam size \(b\), the algorithm starts by querying the model to obtain its predictions of possible subword units, ranked by their probability. The algorithm uses two priority queues: one called candidates which ranks the sequences of subword units that still need to be explored during the search, and one called bestTokens which contains the \(k\) highest probability complete tokens that have been expanded so far. Each candidate is a structure with two fields, text which is the concatenation of all the subword units in the candidate, and prob which is the product of the probabilities of each subword unit in the candidate. Both of the priority queues are sorted by the probability of the candidate.

In each iteration, the algorithm pops the \(b\) best candidates from the candidates queue, expands them with one additional subword unit, and scores their expansions. If an expansion creates a token (the new subword unit ends with \(</t>\)) then it is pushed onto the token queue and the worst token is popped. This maintains the invariant that bestTokens has size \(k\). If the new expansion is not a complete token, then it is pushed onto the candidates queue, where it can potentially be expanded in the next iteration.

### 5.3 Caching

We also implement a simple caching mechanism for our NLM to exploit the locality of source code, particularly previously defined identifiers. At test time, each time an identifier is encountered, the 5-token history that preceded it is added to a cache alongside it. Differently to n-grams, we do not store probabilities, as the NLM will compute them. If the current 5-token history exists in the cache, the identifiers that followed it are retrieved (this is in practice very small, usually 1 or 2 identifiers). These identifiers are then scored by the NLM, and their probabilities are normalized to 1. The beam search described earlier is then run, and the two probability distributions are merged, according to a cache weight parameter: \(\text{cache}_\text{pred} \times \text{cache}_\text{weight} + \text{beam}_\text{pred} \times (1 - \text{cache}_\text{weight})\). The top 10 of the merged predictions are then returned.

We set the cache weight to 0.3. Note that, like beam search, this is a test-time only addition that does not affect training.

### 5.4 Dynamic adaptation to new projects

A global LM, trained in a cross-project setting, will perform better if it is adapted to a new project \([44, 81]\).LMs with n-grams also employ caches for this. Simply training an NLM from scratch on a new project will not have enough data to be effective, while training a new model on both the original training set and the new project would be impractical and computationally expensive.

Instead, we use a simple method of dynamically adapting our global NLMs to a new project. Given a new project, we start with the global NLM and update the model parameters by taking a single gradient step on each encountered sequence in the project after testing on it. This series of updates is equivalent to a single training epoch on the new project. (In our evaluations in Section 6, we will split up the project files in such a way that we are never training on our test set.) We unroll the GRU for 20 time steps instead of 200 as in our global models, in order to update the parameters more frequently. We apply only one update for two reasons. First, it is faster, allowing the model to quickly adapt to new identifiers in the project. Second, taking too many gradient steps over the new project could cause the NLM to give too much weight to the new project, losing information about the large training set.

### 6 Evaluation

#### Intrinsic Evaluation: Language Modeling

A good language model assigns high probabilities to real sentences and low probabilities to wrong ones. For code, fragments that are more likely to occur in human-written code should be assigned higher probability. Precise scoring of code fragments is essential for tasks such as translating a program from one programming language to another \([54, 66]\), code completion \([32, 76]\), and code synthesis from natural language and vice versa \([7, 27, 67, 73]\).

As in previous work, our intrinsic metric is the standard cross entropy. Cross entropy defines a score over a sequence of tokens \(t_1, t_2, \ldots, t_{|Q|}\). For each token \(t_i\), the probability \(p(t_i|t_1, \ldots, t_{i-1})\) of each token is estimated using the model under evaluation. Then the average per token entropy is \(H_p(C) = -\frac{1}{|Q|} \sum_{i=1}^{|Q|} \log p(t_i|t_1, \ldots, t_{i-1})\). Cross entropy is the average number of bits required in every prediction; lower values are better. It not only takes into account the correctness of the predictions, but also rewards high confidence.

Our NLMs define a distribution over subwords, not tokens. To compute cross entropy for subword NLMs, we segment each token \(t_i\) into subwords \(t_i = w_1 \ldots w_{|t_i|}\). Then we compute the product \(p(t_i|t_1, \ldots, t_{i-1}) = \prod_{m=1}^{|t_i|} p(w_m|t_1, \ldots, t_{i-1})\), where the right hand side can be computed by the subword NLM. This probability allows us to compute the cross entropy \(H_p(C)\).

#### Extrinsic Evaluation: Code Completion

We report the performance of our LMs on code completion, which is the task of predicting each token in a test corpus given all the previous tokens in the file. We measure performance with mean reciprocal rank (MRR), as is common in code completion evaluation \([17, 41, 76, 81]\). Each time the LM makes a prediction, we get a ranked list of \(Q\) predictions. For each one, the reciprocal rank is the multiplicative inverse of the rank of the first correct answer. MRR is the average of reciprocal ranks for a sample of queries \(Q\):

\[
\text{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{rank}_i}.
\]

A simplified description of MRR is that it averages top-\(k\) predictive performance across various \(k\). Note that a correct suggestion at rank 1 yields an MRR of 1; at rank 2, 0.5; at rank 10, 0.1. Thus, a
small difference in MRR could indicate a large change in the ranked list, especially for higher MRR values.

**Code Completion Scenarios.** We use three scenarios from previous work [41]: Each static, dynamic, and maintenance settings simulates a different way of incorporating NLMs in an IDE. The task is always to predict test set tokens, but the training sets differ:

*Static tests.* The model is trained on a fixed training corpus, and later evaluated on a separate test dataset. This is a cross-project setting: train, validation, and tests sets all contain separate projects. This simulates a single global LM that is trained on a large corpus of projects and then deployed to clients without adaption.

*Dynamic tests.* In addition to the training set, the model can update its parameters after it has made predictions on files in the test set (it never trains on test data). Our NLMs are adapted using the procedure described in Section 5.4. After each project, we restore the model to the global LM learned from the train set only. This simulates a setting in which some files from the project of interest are available for dynamic adaption.

*Software maintenance tests.* This scenario is even closer to real world usage, simulating everyday development where programmers make small changes to existing code. The LMs are tested on one file at a time in the test set. For each test file $F$, the train set plus all other files in the test project except $F$ is used as training data. As this requires retraining the NLM once per file in the test set, this scenario was previously deemed infeasible for NLMs in [41].

**Identifiers only.** Recent work observed that LMs for completion perform worse on identifiers than other tokens [42]. Therefore, we also report model performance, i.e. entropy and MRR, on identifier tokens only (excluding primitive types). To clarify differences between methods, we also report recall at rank 1 (R@1), the percentage of all identifier usages which are correctly predicted at rank 1, and similarly recall at rank 10 (R@10), the percentage when the correct identifier appears anywhere in the model’s top 10 predictions.

7 RESEARCH QUESTIONS

**RQ1.** How does the performance of subword unit NLMs compare to state-of-the-art LMs for code? We compare subword unit NLMs to standard n-gram LMs [44], cache LMs [81], state-of-the-art n-gram LMs with nested caching [41], token-level NLMs [88], and heuristic splitting NLMs [3]. We do not compare with PHOG [14] and pointer network RNNs [58]: both do not have a full implementation available. We do not evaluate character-level NLMs as they have not shown benefits for NLP.

**RQ2.** Can subword unit NLMs scale to large code corpora? Does the additional training data improve performance? Training on a larger corpus may improve a model’s performance, but adding more data tends to have diminishing returns. After some point, a model’s performance saturates. We evaluate if NLMs can make better use of large corpora than n-gram models. Moreover, training on larger data uses introduces scaling issues. Thus, performance in terms of runtime cost, memory usage, and storage becomes important.

**RQ3.** How does the performance of subword unit NLMs vary across programming languages? In principle the learning methods for NLMs are language agnostic; however, the majority of studies evaluate only on Java. We check if code LMs are equally effective on other programming languages: C’s terseness, or Python’s lack of type information could negatively impact an LM’s performance.

**RQ4.** Is the dynamic updating effective to adapt subword unit NLMs to new projects? New projects introduce many new identifiers that do not appear even in a large cross-project corpus. An n-gram LM can exploit the strong locality that characterises code through caching [44, 81]. Thus we ask whether NLMs can also benefit from dynamic adaptation via the procedure presented in Section 5.4. We compare our dynamic adaptation technique against two dynamic n-gram models: cache LMs [81] and nested cache LMs [41].

**RQ5.** Are NLMs useful beyond code completion? NLMs in NLP have shown to be useful in a variety of tasks, including translation or summarization; they have been recently shown to be state of the art in transfer learning. While testing all of these scenarios vastly exceeds the scope of this paper, we test whether NLMs improve upon n-gram LMs in the task of detecting buggy code [75].

8 RESULTS

Table 2 presents the evaluation metrics of all scenarios; we refer to it continuously. We used the n-gram implementation\(^2\) used in [41] with the same parameters (n = 6); all NLMs are ours. We compute MRR on the first million tokens of the test set, as in [41].

8.1 RQ1. Performance of Models

Because the full data set is so large, we compare the different variants of n-gram models against each other on the small Java training set, and then we compare the best n-gram LM against our BPE NLM on the large Java data set. In Table 2, we see that the nested cache model has the best performance of the n-gram models, with a large improvement over the simpler models (for example, improving MRR from 58% to 77% on Java against the basic n-gram model). This is consistent with the results of [41]. However, our BPE NLM outperforms it. (Note that cache models can not be evaluated in the static scenario since the cache would adapt to the test set). Moving to the large data set, we find that the BPE NLM still outperforms the nested cache model, even though the nested cache model was specifically designed for code. While previous work [42] found that closed NLMs underperformed on identifiers, we find that our BPE NLMs do not. In the dynamic scenario, 74% of identifiers are predicted within the top 10 predictions, with up to nearly 56% in first position.

**Open vs closed vocabulary.** To specifically evaluate the effect of relaxing the closed vocabulary assumption, we compare our open vocabulary NLM to two closed vocabulary NLMs: one that uses full tokens (Closed NLM), and another that splits tokens according to conventions (Heuristic NLM). Those models have otherwise the same architecture as the open vocabulary. In both cases, we find that the open-vocabulary NLM significantly outperforms both closed vocabulary NLMs, and can be trained even in the maintenance setting, unlike the closed versions. Of note, our closed vocabulary NLM performs better than the one in [42], as it utilizes a fully connected hidden layer and dropout. Finally, in Table 3 we report the performance of the open vocabulary NLMs with different

---

\(^2\)A naive approach to the software maintenance scenario retrained the model from scratch for every test file, which was rightly deemed infeasible for NLMs by [41].

\(^3\)https://github.com/SLP-team/SLP-Core, version 0.1

---

---
Table 2: Performance of the various models (bold: best, underlined: second best).

<table>
<thead>
<tr>
<th>MODEL</th>
<th>Java</th>
<th>Java Identifiers</th>
<th>C</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Static Ent MRR Ent MRR</td>
<td>Dynamic Ent MRR Ent MRR</td>
<td>Maintenance Ent MRR Ent MRR</td>
<td>Bugs Ent MRR Ent MRR</td>
</tr>
<tr>
<td>Small Train</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n-gram</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nested Cache</td>
<td>6.02</td>
<td>5.54</td>
<td>5.30</td>
<td>5.82</td>
</tr>
<tr>
<td>Cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nested Cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Closed NLM</td>
<td>4.30</td>
<td>62.28</td>
<td>3.07</td>
<td>71.01</td>
</tr>
<tr>
<td>Heuristic NLM</td>
<td>4.46</td>
<td>50.95</td>
<td>3.34</td>
<td>64.05</td>
</tr>
<tr>
<td>BPE NLM (512)</td>
<td>4.77</td>
<td>63.75</td>
<td>2.54</td>
<td>77.02</td>
</tr>
<tr>
<td>BPE NLM (512) + cache</td>
<td>4.77</td>
<td>64.27</td>
<td>2.08</td>
<td>77.30</td>
</tr>
<tr>
<td>BPE NLM (512) + cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Large Train</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nested Cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BPE NLM (512)</td>
<td>3.15</td>
<td>70.84</td>
<td>1.72</td>
<td>79.94</td>
</tr>
<tr>
<td>BPE NLM (512) + cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BPE NLM (2048)</td>
<td>2.40</td>
<td>75.81</td>
<td>1.23</td>
<td>82.41</td>
</tr>
<tr>
<td>BPE NLM (2048) + cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Effect of vocabulary size on Java performance of our open-vocabulary models (Python and C are similar).

<table>
<thead>
<tr>
<th>VOCAB SIZE</th>
<th>Static Ent MRR Ent MRR</th>
<th>Dynamic Ent MRR Ent MRR</th>
<th>Maint. Ent MRR Ent MRR</th>
<th>Bugs Ent MRR Ent MRR</th>
<th>% Ent ↓</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small Train</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>4.90</td>
<td>62.87</td>
<td>2.33</td>
<td>75.66</td>
<td>1.46</td>
</tr>
<tr>
<td>5000</td>
<td>4.78</td>
<td>63.80</td>
<td>2.27</td>
<td>77.14</td>
<td>1.51</td>
</tr>
<tr>
<td>10000</td>
<td>4.77</td>
<td>63.75</td>
<td>2.54</td>
<td>77.02</td>
<td>1.60</td>
</tr>
<tr>
<td>Large Train</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>3.59</td>
<td>68.87</td>
<td>1.84</td>
<td>77.69</td>
<td>1.03</td>
</tr>
<tr>
<td>5000</td>
<td>3.35</td>
<td>69.87</td>
<td>1.72</td>
<td>79.18</td>
<td>1.06</td>
</tr>
<tr>
<td>10000</td>
<td>3.15</td>
<td>70.84</td>
<td>1.72</td>
<td>79.94</td>
<td>1.04</td>
</tr>
</tbody>
</table>

vocabulary sizes, obtained after 2000, 5000, and 10000 BPE merge operations. We see that performance on the small training set is similar across vocabulary sizes: a large vocabulary is not required for good performance.

Caches, and larger capacity: Both our cache and increasing model capacity (from 512 to 2048 features) are beneficial, particularly for the identifiers. The cache improves MRR by 3 to 4%, with more improvements for low ranks, which is especially important for completion. On the small corpus, the large model improves MRR by nearly 3%, a smaller improvement than adding the cache. Both improvements are complementary, increasing identifier MRR by close to 6%.

8.2 RQ2. Large Corpora

We contrast performance between small and large training sets.

Leveraging data. When trained on larger corpora, the performance of n-gram models (including nested cache variants) gets saturated and they are unable to effectively leverage the extra information [41]. In contrast, our model can better leverage the increase in training data when trained on the full corpus. In the static scenario, our NLMs decrease entropy by about 1.5 bits, while MRR increases by about 6%. More data helps our NLMs learn to synthesize identifiers from subwords better and with higher confidence.

The improvements are smaller but still exist when the NLMs use dynamic adaptation: for all encoding sizes the entropy improves by 0.5 bits and MRR by 2 to 3%. In contrast, the nested cache n-gram model entropy improves by less than 0.1 bits and MRR by less than 0.4%. From that we conclude that subword unit NLMs can utilize a large code corpus better than n-gram models. As shown in Table 3, larger training corpora tend to favor NLMs with larger vocabularies, particularly in terms of MRR; larger models leverage the additional data even better. For all models, the improvements are more visible for identifiers: the large train alone contributes close to 7% of MRR for identifiers, versus 2% overall for the NLM.

Finally, larger NLMs (2048 features) are even better at leveraging the additional data even better. For all encoding sizes the entropy improves by less than 0.1 bits and MRR by less than 0.4%. From that we conclude that subword unit NLMs can utilize a large code corpus better than n-gram models. As shown in Table 3, larger training corpora tend to favor NLMs with larger vocabularies, particularly in terms of MRR; larger models leverage the additional data even better. For all models, the improvements are more visible for identifiers: the large train alone contributes close to 7% of MRR for identifiers, versus 2% overall for the NLM. Finally, larger NLMs (2048 features) are even better at leveraging the additional training data, due to their increased capacity. Similarly, the cache still improves performance further, even with the large training set; both improvements complement each other.

Resource usage. While the nested cache n-gram model is competitive with Java identifiers, this comes at a significant cost: resource usage. Disk usage for n-gram models range from 150 to 500 Mb in the small training set to 6 to 8.5Gb in the large training set. RAM usage is even more problematic, as it ranges from around 5Gb in the small training set, up to 50 to 60Gb in the large training set.
This makes the large n-gram models unusable in practice as they exceed the memory requirements of most machines.

In contrast, the NLMs do not vary significantly with training set size; their size is fixed. They range from 15MB (BPE 2K) to 45MB (BPE 10K) on disk (up to 240MB for the large capacity models). RAM usage for NLMs vary between 2 to 4GB when training (and can be reduced at the expense of speed by reducing batch size), and is considerably lower at inference time (for actual code completion), ranging from 250 to 400MB. Thus, if we compare practically applicable models, the small NLM outperforms the small nested cache n-gram model by up to 5.13% in identifier MRR, and up to 5.75% recall at 1; the large NLM does so by 8.68% (MRR), and 10.81% (recall at 1).

The open vocabulary makes training NLMs on large corpora scalable as vocabulary ceases to grow with corpus size; training time scales linearly with added data. Our largest NLM (BPE 10k, 2048 features), can process around 350 to 550 hundred thousand tokens per minute (roughly 100 to 300 projects per hour depending on project size) on a consumer-grade GPU. This makes our dynamic adaptation procedure, which trains one project for one epoch, clearly feasible. Training the initial model is still a large upfront cost, but it takes from a day (small NLM) up to two weeks (large NLM) on our largest dataset, and needs to be performed once. At inference time, predicting 10 tokens with beam search takes a fraction of a second, fast enough for actual use in an IDE, even without additional optimization. This is not true for the closed models.

Open-vocabulary NLMs can scale; furthermore, they leverage the increased training data effectively. Large n-gram models do not scale in terms of resources.

8.3 RQ3. Multiple Languages

We contrast Java performance with Python and C. We see interesting differences between Java, Python, and C. First, n-gram models perform considerably worse in Python, while NLMs do very well.

We hypothesize that this is due to the smaller size of Python projects in our corpus, which reduces opportunity for caching (the average Python project is 2 to 3 times smaller than the average Java project).

C projects, on the other hand, are competitive with Java projects, particularly with caching; they are on average 2 times larger. Interestingly, the nested and nested cache n-gram models perform comparatively worse in C than in Java: C projects tend to have a flatter structure, rendering the nesting assumption less effective in this case. Finally, the (not applicable in practice) large n-gram model outperforms our NLMs for C. We observed anecdotal evidence that there is considerable duplication in the C corpus, which may affect this result [1]. For NLMs, the performance is more even across the board, with overall slightly worse performance for C, and somewhat better performance for Python.

Our NLM performance results hold for Java, C, and Python.

8.4 RQ4. Dynamic Adaptation

We evaluate the effectiveness of our proposed method for adaption of NLMs in the dynamic and maintenance scenarios. This is crucial for practical usage of NLMs, because the dynamic and maintenance scenarios simulate the setting where the developer is modifying a large, existing project. Using within-project data provides a large performance boost: Even though within each scenario, our NLMs outperform n-grams, most n-gram models in the dynamic scenario outperform NLMs in the static scenario. The improvement due to dynamic adaptation is greater than the improvement due to an NLM. Of note, the situation in the large training set is different: the static large NLM trained on the large training set outperforms the cache n-gram LMs in the dynamic scenario, and is competitive with it in the maintenance scenario, in other words, our large data set is so large that it almost makes up for not having within-project data, but within-project information is clearly still crucial.

Once we apply the dynamic adaptation method to the NLMs, the picture changes. With dynamic adaptation, our model achieves even better cross-entropy than the current state-of-the-art [41], making it an effective technique to fine-tune an NLM on a specific project. Using this method, it is even possible to evaluate NLMs on the maintenance scenario, which was previously deemed infeasible by [41] since multiple models had to be created, each trained on the entirety of the test set minus one file. This is possible for us because the combination of a small vocabulary size and our finetuning method running for only one epoch make this scenario much faster.

Open vs closed NLMs. Interestingly, the difference in performance between the open and closed vocabulary NLMs is larger in the dynamic setting. We hypothesize that dynamic adaptation helps the open-vocabulary model to learn project-specific patterns about OOV words; this is not possible for a closed vocabulary NLM.

Dynamic adaptation for NLMs yields the state of the art; static NLMs are competitive with some dynamic n-gram models, which bodes well for transfer learning.

8.5 RQ5. Bug Detection

Previous work has observed that n-gram language models can detect defects as they are less “natural” than correct code [75]. In short, defective lines of code have a higher cross-entropy than their correct counterparts. To assess whether our code NLM is applicable beyond code completion, we compare the ability of different language models to differentiate between the two on the well-known Defects4J dataset [53]. Defects4J contains 357 real-world defects from 5 systems. Both a buggy and a corrected version of the system are provided and the changed lines can be extracted. We compute the difference in entropy between the buggy and the fixed version for each of the diff patches provided. The extracted code snippets usually contains a few unchanged surrounding lines that provide useful context for the LMs. We expect a better LM to have a larger entropy difference between the defective and the corrected version.

We compute these metrics only for LMs in a static setting for three reasons: 1) we simulated the setting in which a bug detector is trained on one set of projects and used on unseen ones, 2) it is not clear how caches would be used in this scenario (should the LM “know” which file a bug is in?), and 3) doing so could involve training two LMs for each defect, which is very expensive.

The results are shown in the Java “bugs” column in Tables 2 and 3. As we hypothesized, open vocabulary NLMs feature a larger entropy drop for clean files than n-gram LMs or closed NLMs. The
9 CONCLUSIONS

Source code has a critical difference with natural language: developers can arbitrarily create new words, greatly increasing vocabulary. This is a great obstacle for closed-vocabulary NLMs, which do not scale to large source code corpora. We first extensively studied vocabulary modelling choices, and showed that the only viable option is an open-vocabulary NLM; all other vocabulary choices result in large vocabularies, high OOV rates, and rare words.

We then presented a new open-vocabulary NLM for source code. By defining the model on subword units, which are character sub-sequences of tokens, the model is able to handle identifiers unseen in training while shrinking vocabulary by three orders of magnitude. As a consequence, our NLM can scale to very large corpora: we trained it on data sets over a hundred times larger than had been used for previous code NLMs. Our NLM also uses beam search, dynamic adaptation, and caching to efficiently generate tokens and adapt to new projects. Finally, we showed that our NLM outperforms recent state-of-the-art models based on adding nested caches to n-gram language models for code completion and bug detection tasks, in a variety of scenarios, and in three programming languages.

Of course, this study has limitations: While we tried to be exhaustive and evaluated a large number of scenarios, we could not evaluate all the possible combinations (hundreds) due to the resources needed, such as some large models or some large training scenarios. For this reason, we also refrained to evaluate other NLM architectures such as LSTMs [45], QRNNs [16], Transformers [85], or additional neural cache variants [62, 86]. For the same reason, as in [41] we also limited MRR to 1 million tokens, which may cause discrepancies with entropy metrics as they are not evaluated on the same test set. We also limited ourselves to three languages, and did not fully evaluate the impact of code duplication [1].

We also hope that the simplicity and scalability will enable large capacity models for code, and the transfer learning opportunities they bring [28, 72]: this has been explored in software engineering, albeit not for source code [77]. Improved language models for code have the potential to enable new tools for aiding code readability [2], program repair [13, 18, 40, 75], program synthesis [39] and translation between programming languages [54, 66]. Finally, the technique of using subword units is not limited to language modeling, but can easily be incorporated into any neural model of code, such as models to suggest readable names [3], summarizing source code [5, 49], predicting bugs [70], detecting code clones [87], comment generation [47], and variable de-obfuscation [10].

Open-vocabulary NLM are better bug detectors than n-gram LMs, particularly when trained on large corpora.

10 ARTIFACTS

Several artifacts were used to conduct this study: data, source code, and models. To improve replication of this work, the specific version of each artifact used in this study can be referenced via a DOI. Table 4 lists the DOI of each artifact. This paper can be referenced when any of these artifacts is used.

Datasets. The datasets described in 3 were published in previous work: The Java corpus was produced by Allamanis et al. [6], and also used in [41]. The C corpus was mined in [29] and the Python corpus was mined in [31]. We use the raw datasets for the vocabulary study, but preprocess them for NLM training. Further, we defined training and test sets for the C and Python corpora, and defined the large training set for the Java corpus. Source code. We implemented the codeprep library that supports a variety of pre-processing options for source code. We used codeprep to gather the vocabulary statistics presented in section 4. Researchers that wish to use the library to pre-process source code for their own study can find the library at: https://github.com/giganticode/codeprep.

The open vocabulary language model described in 5, as well as the scripts implementing the training procedure and the evaluation scenarios are available in the OpenVocabCodeNLM library. Researchers wishing to extend our model can find it on GitHub at: https://github.com/mast-group/OpenVocabCodeNLM.

Models. The models that were trained and evaluated in section 8 are also made available for further use. Each model was trained on GPUs for periods ranging from a few hours, up to two weeks. These models can be used as-is for inference in a code completion scenario. Alternatively, they may be fine-tuned for other tasks, such as classification [46, 77].

11 ACKNOWLEDGEMENTS

This work was supported in part by the EPSRC Centre for Doctoral Training in Data Science, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L016427/1) and the University of Edinburgh. This work was partially funded by the IDEALS and ADVERB projects, funded by the Free University of Bozen-Bolzano. Parts of the results of this work were computed on the Vienna Scientific Cluster (VSC).


