Maybe Deep Neural Networks are the Best Choice for Modeling Source Code

ABSTRACT

Statistical language modeling techniques have successfully been applied to source code, yielding a variety of new software development tools, such as tools for code suggestion and improving readability. A major issue with these techniques is that code introduces new vocabulary at a far higher rate than natural language, as new identifier names proliferate. But traditional language models limit the vocabulary to a fixed set of common words. For code, this strong assumption has been shown to have a significant negative effect on predictive performance. But the open vocabulary version of the neural network language models for code have not been introduced in the literature. We present a new open-vocabulary neural language model for code that is not limited to a fixed vocabulary of identifier names. We employ a segmentation into subword units, subsequences of tokens chosen based on a compression criterion, following previous work in machine translation. Our network achieves best in class performance, outperforming even the state-of-the-art methods of Hellendoorn and Devanbu that are designed specifically to model code. Furthermore, we present a simple method for dynamically adapting the model to a new test project, resulting in increased performance. We showcase our methodology on code corpora in three different languages of over a billion tokens each, hundreds of times larger than in previous work. To our knowledge, this is the largest neural language model for code that has been reported.

统计语言建模技术已成功地应用于源代码，产生了各种新的软件开发工具，如代码建议和提高可读性的工具。这些技术的一个主要问题是，随着新标识符名称的激增，代码引入新词汇表的速度远远高于自然语言。但是传统的语言模型将词汇限制在一组固定的常用词上。对于代码，这种强假设已被证明对预测性能有显著的负面影响。但是，开放词汇版本的神经网络语言模型的代码还没有在文献中介绍。提出了一种新的开放词汇神经语言模型，该模型不局限于固定的标识符名称词汇。我们在机器翻译的前面工作的基础上，将符号分割成子单词单元，即根据压缩标准选择的符号的子序列。

1 INTRODUCTION

Large corpora of open source software projects present an opportunity for creating new software development tools based on machine learning [4].Many of these methods are based on the hypothesis that much of software is natural, that is, because software is written for humans to read, it displays some of the same statistical properties as natural language. To quantify the degree of naturalness of a piece of software, Hindle et al [40] propose the use of statistical language modeling. A language model is a probability distribution over strings; by training a language model (LM) on a large corpus of well-written code, we hope that the LM will assign high probability to new code that is similar to the training set, in other words, code that is well-written, easy to read, and natural. There is now a large literature on language modeling for code[6, 12, 22, 37, 40, 58, 69].

4是大型survey:

[4] Miltiadis Allamanis, Earl T. Barr, PremkumarDevanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. Comput. Surveys (2018).

40 [40] Abram Hindle, Earl T. Barr,Zhendong Su,MarkGabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In Proceedings of the 34th International Conference on Software Engineering (ICSE ’12). IEEE Press, Piscataway, NJ, USA, 837–847. http://dl.acm.org/citation.cfm?id=2337223.2337322

大型开源软件项目库为创建基于机器学习[4]的新软件开发工具提供了机会。这些方法中的许多都是基于这样一个假设，即大部分软件都是自然的，也就是说，因为软件是为人类阅读而编写的，所以它显示了一些与自然语言相同的统计特性。为了量化软件的自然程度，Hindle等人提出使用统计语言建模。语言模型是字符串上的概率分布;通过在编写良好的代码的大型语料库上训练语言模型(LM)，我们希望LM将高概率分配给与训练集类似的新代码，换句话说，就是编写良好、易于阅读和自然的代码。现在有大量关于代码语言建模的文献[6、12、22、37、40、58、69]。

[6] Miltiadis Allamanis and Charles Sutton. 2013. Mining Source Code Repositories at Massive Scale Using Language Modeling. In Proceedings of the 10th Working Conference onMining Software Repositories (MSR ’13). IEEE Press, Piscataway, NJ, USA, 207–216. http://dl.acm.org/citation.cfm?id=2487085.2487127

[12] Pavol Bielik, INF ETHZ, Veselin Raychev, andMartin Vechev. 2016. PHOG: Probabilistic Model for Code. Proceedings of the 33rd International Conference on Machine Learning (ICML-16) (2016).

[22] Hoa Khanh Dam, Truyen Tran, and Trang Pham. 2016. A deep language model for software code. arXiv preprint arXiv:1608.02715 (2016).

[37] Vincent J. Hellendoorn and Premkumar Devanbu. 2017. Are Deep Neural Networks the Best Choice for Modeling Source Code?. In Proceedings of the 2017 11th JointMeeting on Foundations of Software Engineering (ESEC/FSE 2017). ACM, New York, NY, USA, 763–773. https://doi.org/10.1145/3106237.3106290

[58] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2013. A Statistical Semantic Language Model for Source Code. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013). ACM, New York, NY, USA, 532–542. https://doi.org/10.1145/2491411.2491458

[69] Zhaopeng Tu, Zhendong Su, and Premkumar T. Devanbu. 2014. On the localness of software. In Proceedings of the 22nd Symposium on the Foundations of Software Engineering. ACM, 269–280. https://doi.org/10.1145/2635868.2635875

Such models have enabled research on a broad suite of new software engineering tools. For example, the ability to automatically quantify the naturalness of software has enabled new tools for autocompletion [40, 64], improving code readability [2, 63], and program repair [62]. Furthermore, recent work in natural language processing (NLP) [24, 59] has shown that LMs and sequence models in general learn useful word embeddings, which can then be used for downstream tasks in the same way as older, word2vec style embeddings [54]. Such continuous embeddings have formed the foundation of important software engineering tools. Examples of such are suggesting readable function and class names [3], summarizing source code [5, 43], predicting bugs [60], detecting code clones [73], comment generation [42], fixing syntactic errors [47], and variable de-obfuscation [9]. Therefore, improved LMs for code have the potential to enable improvements in a diverse variety of software engineering tools.

这些模型使得能够研究一系列新的软件工程工具。 例如，自动量化软件自然性的能力已经启用了新的自动完成工具[40,64]，提高了代码可读性[2,63]和程序修复[62]。 此外，最近在自然语言处理（NLP）[24,59]中的工作表明LM和序列模型通常学习有用的词嵌入，然后可以像旧的word2vec样式嵌入一样用于下游任务[54]]。 这种连续嵌入已经形成了重要的软件工程工具的基础。 这样的例子是建议可读函数和类名[3]，总结源代码[5,43]，预测错误[60]，检测代码克隆[73]，评论生成[42]，修复语法错误[47]，以及 变量去混淆[9]。 因此，改进的代码LMs有可能改进各种各样的软件工程工具。

However, a provocative recent paper by Hellendoorn and Devanbu [37] argues that there is a key general challenge in deep learning models of code, which significantly hinders their usefulness.

This is the out of vocabulary (OOV) problem, which is that new identifier names are continuously invented by developers [6]. These out-of-vocabulary (OOV) tokens cannot be predicted by a language model with a fixed vocabulary, because they have not occurred in the training set. Although this problem also exists in natural language text, it is much more severe in code. For instance, the vocabulary size of 74046 tokens used in [37] covers only about 85% of identifier occurrences and 20% of distinct identifiers that appear in the test set. Therefore, standard methods from NLP of dealing with this problem, such as introducing a single special token to represent OOV tokens [17], still fall short of fully addressing the OOV problem for code. Instead, Hellendoorn and Devanbu [37] present a n-gram LM with several code-specific extensions. This enhanced n-gram model shows improved performance over an off-the-shelf neural language model for code, a surprising result because for natural language, neural models consistently outperform n-gram models. Based on these improvements, Hellendoorn and Devanbu raise the provocative suggestion that deep models might not be the best choice for modeling source code.

In this paper, we argue that to the contrary, perhaps deep networks are a good choice for modeling source code, because it is possible to overcome the limitations highlighted by [37]. More specifically, we address the key challenge of deep models of source code that was highlighted in the previous work, namely, the OOV problem,

by introducing an open-vocabulary neural language model for source code. An open vocabulary model is not restricted to a fixed sized vocabulary determined at training time; for example, some types of open vocabulary models predict novel tokens character by character. Our open vocabulary model is based on the idea of subword units, following previous work from neural machine translation [65]. Subword units are a way of combining the strengths of character-level and token-level models. Each 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. Including all single characters as subword units will allow the model to predict all possible tokens, so there is no need for special OOV handling. The vocabulary of subword units in the model is inferred from the training set using a compression-based heuristic called byte pair encoding (BPE).

The use of subword unit NLMs has two main advantages for code: First, even for an OOV token t that has never occurred in the training data, its subword units will have occurred in training, so the model can predict what will follow t based on what tended to follow its subword units in the training data. Second, training of a subword unit NLM on large corpora corpora is much faster than a token level model, as a relatively smaller number of subword units can still lead to good performance. On large opensource corpora, we show that our model is indeed more effective for code language modeling than previous n-gram or neural models.

On Java, our model is able to achieve a predictive performance of 3.15 bits per token and 70.84% MRR in a cross project evaluation. Simultaneously achieving predictive performance of 1.04 bits per token and 81.16% MRR in a within project setting, This is the best performance that we are aware of in the literature for a single model.

In particular, our contributions can be summarized as follows:

• We present the first open-vocabulary neural language model (NLM) for source code. This is based on an automatic segmentation of code tokens into smaller subword units that learns the common statistical internal patterns within identifier names.

• We show that our model outperforms previous neural and n-gram LMs for code across three programming languages:

Java, C, and Python. For C and Python, we are the first to showcase deep learning results for corpora of our size. Interestingly, we show that while the n-gram LMs are unable to improve their performance on larger training sets, neural LMs are able to improve their performance given more data.

• Ours is the largest neural LM for code reported in the literature, trained on 1.7 billion tokens, which is 107 times larger than in previous work.

• We present a simple heuristic to adapt a NLM trained on a large corpus of projects to a single project of interest, an important scenario for practical development tools. This allows us to report results on the code maintenance scenario of [37], which was previously infeasible for NLMs.

2 RELATED WORK

**Language modeling for code.**

Numerous researchers have applied language modeling techniques on code corpora. These studies are based on the assumption that software is characterized by similar statistical properties to natural language; viz., the naturalness hypothesis [40]. It should be expected that software is repetitive and predictable. Indeed this successfully verified in [30]. Hindle et al. [40] introduced the naturalness hypothesis and showcased that Java code is actually less entropic than a natural language (English).

Nguyen et al. [58] augmented n-gram LMs with semantic information such as the role of a token in the program, e.g., variable, operator, etc. On the other hand, other researchers attempted to exploit the localness of code by augmenting n-gram models with a cache component [69]. The cache contains n-grams that previously appeared in the current file or in nearby ones, such as files from the same package. The model was shown to offer lower entropies for code but not for English. Later, Hellendoorn and Devanbu [37] extended this idea to nested scopes, outperforming vanilla NLMs and achieving best-in-class performance on a Java corpus.

许多研究人员已经在代码语料库中应用了语言建模技术。这些研究是基于软件具有与自然语言相似的统计特性的假设;即自然假设[40]。应该预期软件是重复的和可预测的。事实上，这在[30]中成功验证了。 Hindle等。 [40]引入了自然性假设，并展示了Java代码实际上比自然语言（英语）更少熵。

Nguyen等。 [58]用语义信息增加n-gram LM，例如令牌在程序中的作用，例如变量，运算符等。另一方面，其他研究人员试图通过增加n-gram模型来利用代码的局部性使用缓存组件[69]。缓存包含先前出现在当前文件或附近文件中的n-gram，例如来自同一包的文件。该模型显示为代码提供较低的熵，但不提供英语。后来，Hellendoorn和Devanbu [37]将这个想法扩展到嵌套范围，超越了vanilla NLM并在Java语料库上实现了最佳性能。

Beyond n-gram models, several other types of methods have been employed to model code. A generative model for code called probabilistic higher order grammar (PHOG)was introduced by [12], which generalizes probabilistic context free grammars [44]. Also, both simple RNNs were used in [74] and LSTMs [22] to learn an NLM for code. The LSTMs were shown to perform much better than simple RNNs. This conclusion was later confirmed by Hellendoorn and Devanbu [37]. Lastly, [48] attempt to improve code completion performance on OOV words by augmenting an RNN with a pointer network [72]. Their pointer network learns a copy mechanism over the current context using attention that is useful in code completion. Once an OOV token has been used once, the copy mechanism can learn to re-use it, but unlike our model it cannot predict its first usage and it is not designed to learn dependencies between the OOV token and the next tokens as it learns no representations for OOV words, in contrast to our method. That said, the subword units learned by our model can be used by any sort of neural model for code, so it could be fruitful in future work to combine our approach with this previous work, for example by augmenting our network with a pointer network component.

除了n-gram模型之外，还采用了其他几种方法来对代码进行建模。 [12]引入了一种称为概率高阶语法（PHOG）的代码生成模型，它概括了概率上下文无关语法[44]。此外，在[74]和LSTM [22]中使用了两个简单的RNN来学习代码的NLM。 LSTM显示出比简单RNN更好的性能。 Hellendoorn和Devanbu [37]后来证实了这一结论。最后，[48]尝试通过用指针网络扩充RNN来提高OOV字的代码完成性能[72]。他们的指针网络使用在代码完成中有用的注意力来学习当前上下文的复制机制。一旦OOV令牌被使用一次，复制机制就可以学习重用它，但与我们的模型不同，它无法预测它的第一次使用，并且它不是为了学习OOV令牌和下一个令牌之间的依赖关系而学习的与我们的方法相比，OOV词的表示。也就是说，由我们的模型学习的子词单元可以被任何类型的神经模型用于代码，因此在将来的工作中将我们的方法与之前的工作相结合可能是富有成效的，例如通过使用指针网络组件来扩充我们的网络。

**Applications of code language models.**

Probabilistic code models have enabled many applications in software engineering. One example is recommender systems aiming to aid developers in writing or maintaining code [51]. Hindle et al. used a token-level LM for code completion [40], while later, Franks et al. [29] improved on performance using the cache n-gram from [69] and built a code suggestion tool for Eclipse [32]. Another application are recommendation systems for variable, method, and class names suggestion [2, 3, 5] that employ relevant code tokens as the LM context.

Campbell et al. [15] used n-gram language models to detect syntax error locations in Java code. Lastly, Ray et al. [62] showcased that buggy code has on average lower probability than correct one and that LMs can spot defects as effectively as popular tools like FindBugs.

概率代码模型在软件工程中应用广泛。一个例子是旨在帮助开发人员编写或维护代码[51]的推荐系统。Hindle等人使用令牌级别LM来完成代码[40]，而稍后，Franks等人使用缓存n-gram对[29]的性能进行了改进[69]，并为Eclipse[32]构建了一个代码建议工具。另一个应用程序是变量、方法和类名建议的推荐系统[2,3,5]，它们使用相关的代码标记作为LM上下文。

Campbell等人使用n-gram语言模型来检测Java代码中的语法错误位置。最后，Ray等人[62]展示了错误代码的平均概率比正确代码低，并且LMs可以像FindBugs等流行工具一样有效地发现缺陷。

Out-of-vocabulary problem.

Even after an LM has been trained on a large corpus of projects, many identifier names are still encountered which are out of vocabulary (OOV) token that have not been previously encountered in the training set. These are called neologisms by [3]. Indeed on our corpora we find that OOV words are many times more common in code than in natural language. Traditional LMs are closed vocabulary, meaning that the vocabulary is fixed based on a training corpus, and they are unable to predict OOV words, which is an obvious issue for code. Apart from that, many identifiers appearing in the training set are rare. This sparsity could potentially confuse the model or result in slower estimation of parameters.

To combat this problem, previous research in language modeling for code has segmented identifiers via a heuristic [3], which splits them on camel case and underscores. Even though the resulting segmentation has the ability to handle some neologisms, it is limited to only combinations of subtokens appearing in the training set and thus unable to achieve an open vocabulary. Additionally, many of these subtokens are still infrequent, which hinders the model’s ability to assign high scores to their compositions. A separate line of work is that several studies have empirically compared different techniques for automatically splitting identifiers [27, 39]. However, this work considers a different problem than us.

That previous work focuses on splitting identifiers into words in a way that matches human judgements. Although our approach also performs identifier splitting, our goal is to split identifiers in a way that improves a language model; in many cases, our subword units will be sequences of characters that are not words. In our application, it is of no interest whether our subword units correspond to words that humans recognize, because our subword units can be trivially reassembled into complete tokens before they are shown to a developer.

即使LM在一个大型的项目语料库上进行了训练，仍然会遇到许多以前在训练集中没有遇到过的词汇表(OOV)令牌之外的标识符名称。这些名称被[3]称为新词。事实上，在我们的语料库中，我们发现OOV单词在代码中比在自然语言中常见许多倍。传统的LMs是封闭词汇，即基于训练语料库的词汇是固定的，无法预测OOV单词，这对于代码来说是一个明显的问题。除此之外，训练集中出现的标识符很少。这种稀疏性可能会混淆模型，或者导致对参数的估计变慢。

为了解决这一问题，以前的代码语言建模研究已经通过启发式[3]对标识符进行了分段，它在驼峰大小写和下划线上对标识符进行了分割。尽管最终的分割能够处理一些新词，但它仅限于训练集中出现的子标记的组合，因此无法实现开放词汇表。此外，许多子标记仍然不频繁，这阻碍了模型为它们的作文分配高分的能力。另一项独立的工作是，一些研究对自动分割标识符的不同技术进行了经验上的比较[27,39]。然而，这项工作考虑的问题与我们不同。

之前的工作重点是将标识符以匹配人类判断的方式拆分为单词。虽然我们的方法也执行标识符拆分，但我们的目标是以改进语言模型的方式拆分标识符;在许多情况下，我们的子单词单位将是非单词的字符序列。在我们的应用程序中，我们的子单词单元是否与人类识别的单词相对应并不重要，因为我们的子单词单元可以在显示给开发人员之前被简单地重新组装成完整的令牌。

**Open vocabulary language models.**

The occurrence of OOV entries in test data is not a code specific problem but has also been an in issue in NLP, especially in morphologically-rich languages, for example. Alternative models have been proposed where the vocabulary is open. Character language models are one such solution where each word is represented as a sequence of its characters.

While recurrent NLMs have shown excellent performances for character-level language modeling [38, 68], the performance of such models is usually worse than those built on the word level [55]. In order for character model to capture the same dependencies as a token level one, the length of the sequences that gradients are calculated upon needs to be increased. This can cause training to become more difficult due to the vanishing or exploding

gradient problems, even for LSTM [41] and GRU [18] recurrent networks which are designed specifically to address gradient problems.

测试数据中OOV条目的出现并不是一个特定于代码的问题，但在NLP中也是一个问题，特别是在形态学丰富的语言中。已经提出了词汇表是开放的替代模型。字符语言模型就是这样一种解决方案，其中每个单词都表示为它的字符序列。

虽然循环NLMs在字符级语言建模方面表现出了优异的性能[38,68]，但这种模型的性能通常比构建在单词级[55]上的模型要差。为了让字符模型捕获与令牌级别1相同的依赖关系，需要增加梯度计算所依据的序列的长度。这可能导致训练变得更加困难，由于消失或爆炸

梯度问题，甚至对于LSTM[41]和GRU[18]循环网络，这是专门为解决梯度问题而设计的。

Another option that attempts to open the vocabulary is to represent words as a sequence of segments that when merged result to the original word. For example, language models have been proposed at the morpheme level[49], the phone level [10], and the syllable level [55]. Other models combine character level models with a caching mechanism to reuse generated tokens [20, 46]. Finally, other researchers have attempted to also learn the segmentation on a text corpus for machine translation [65], which is the approach that we build on in our work. Subword unit segmentations have been used to capture morphology [70] in a language modeling task but the network’s output were on the word level and thus unable output OOV entries. Simultaneously with our work, [52] independently developed a recent LM based on subword units for natural language, and found that it had close to state-of-the-art performance.

尝试打开词汇表的另一个选项是将单词表示为一组片段序列，这些片段在合并到原始单词时生成。例如，在语素水平[49]、电话水平[10]和音节水平[55]提出了语言模型。其他模型结合字符级模型和缓存机制来重用生成的令牌[20,46]。最后，其他研究者也尝试学习机器翻译文本语料库的分割[65]，这也是我们在工作中所建立的方法。在语言建模任务中，子单词单元分段被用来捕获形态学[70]，但是网络的输出在单词级，因此无法输出OOV条目。与此同时，[52]独立开发了一款基于自然语言子词单元的LM，并发现其性能接近于最先进水平。

3 METHODOLOGY

In this section, we present our neural LM for code based on subword units. We begin by giving a brief background on the neural language models we use (Section 3.1). Then we describe how we construct the vocabulary of subword units (Section 3.2), followed by a search procedure for producing k-best completions (Section 3.3), and finally we discuss how we adapt the model on a new project (Section 3.4).

在本节中，我们将介绍基于子单词单元的代码的神经LM。我们首先简要介绍一下我们使用的神经语言模型(3.1节)。然后，我们描述了如何构造子单词单元的词汇表(第3.2节)，然后是一个生成k-best补全的搜索过程(第3.3节)，最后我们讨论了如何在一个新项目中调整该模型(第3.4节)。

3.1 Neural Language Model with GRU Cell

State-of-the-art LMs for natural language are currently based on recurrent neural networks (RNNs) [50, 53, 67]. RNN language models 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, such as long short-term memory (LSTM) units [41] and gated recurrent units (GRUs) [19], have been found to outperform other methods for language modeling [22, 67]. Intuitively, the advantage of an RNN over older language models, such as n-gram language models, is that an n-gram model uses only a short window of n tokens to predict the next token, whereas an RNN can potentially take into account the entire previous history of the sequence. The advantage of gated units are that the gates allow the network to learn when to forget information from the hidden state and take newer, more important information into account [41]. Among different kinds of gated units, GRUs have been shown to perform comparably to LSTMs across different applications [21]. In our initial experiments we found GRUs to slightly outperform LSTMs when trained on the Java corpus, so we use them in our model.

目前最先进的自然语言LMs基于递归神经网络(RNNs)[50,53,67]。RNN语言模型每次扫描一个输入序列前向一个令牌，预测给定所有前一个令牌的每个令牌上的分布。具有门控单元的神经网络，如长短时记忆(LSTM)单元[41]和门控递归单元[19]，已经被发现在语言建模方面优于其他方法[22,67]。直观地说，相对于较老的语言模型(如n-gram语言模型)，RNN的优势在于，n-gram模型仅使用一个由n个令牌组成的短窗口来预测下一个令牌，而RNN可以潜在地考虑序列的整个先前历史。门控单元的优点是，门允许网络学习何时忘记隐藏状态的信息，并将更新、更重要的信息考虑到[41]中。在不同类型的门控单元中，GRUs在不同应用程序[21]上的性能与LSTMs相当。在我们最初的实验中，我们发现GRUs在Java语料库上的表现略优于LSTMs，所以我们在模型中使用了它们。

Our model is a single layer GRU NLM built upon subword units which have been learned from BPE as described in Section 3.2. 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 [66] and a maximum of 50 training iterations using stochastic gradient descent with a minibatch of 32 for the small training sets and a minibatch size of 64 for the full training sets. After each iteration we tested the network on a validation set and measured its cross entropy (see Section 5.1). If the cross entropy is larger than the previous epoch then we halve the learning rate and this can happen for a maximum of 4 times, otherwise training stops. During training of the global model we unroll the GRU for 200 timesteps. Our implementation is open source, written in Tensorflow [1] and it is available in a public GitHub repository.

我们的模型是一个单层的GRU NLM，建立在从BPE中学习的子单词单元的基础上，如第3.2节所述。对于每个词汇表条目，我们学习了512个特征的连续表示，而GRU状态的大小是相同的。在我们所有的实验中，我们使用了0.1的学习率，0.5的辍学率[66]，并且使用随机梯度下降法进行了最多50次训练迭代，小训练集的最小批量为32次，全训练集的最小批量为64次。每次迭代之后，我们在验证集上测试网络并测量其交叉熵(见5.1节)。如果交叉熵比前一个历元大，那么我们将学习速度减半，这种情况最多可以发生4次，否则训练将停止。在全球模型的训练中，我们将GRU展开200个时间步长。我们的实现是开源的，用Tensorflow[1]编写，可以在公共GitHub存储库中使用。

3.2 Selecting Subword Units Using Byte Pair Encoding

Traditional language models in NLP most commonly operate at the token level [22, 67], meaning that the RNN predicts one token at a time. But for code, this strategy leads to large vocabulary sizes, because identifiers in programming languages often correspond to entire phrases in natural language. Because the number of unique identifiers increases with the size of the corpus [6], this problem makes it infeasible to train code LMs on large corpora. As we later illustrate in Section 6 the vocabulary of three different giga-token code corpora is an order larger than an equivalent English one.

NLP中的传统语言模型通常在令牌级别操作[22,67]，这意味着RNN每次预测一个令牌。但是对于代码来说，这种策略会导致较大的词汇量，因为编程语言中的标识符通常对应于自然语言中的整个短语。由于惟一标识符的数量随着语料库[6]的大小而增加，因此在大型语料库上训练LMs代码是不可行的。正如我们稍后在第6节中所演示的，三个不同的giga-token代码语料库的词汇表的顺序大于相同的英语语料库。

In our code LM, we address this problem by having the model predict subword units rather than full tokens at each time step of the RNN. A subword unit is an n-gram of characters that appear as a subsequence of some token in the corpus. An example of a Java source file segmented into subword units is shown in Figure 1. Notice that we include a special subword unit </t> that marks the end of a token, allowing us to convert from a sequence of subword units back into a sequence of tokens. The subword units in the model are chosen adaptively based on statistical frequency, as we will describe shortly. The effect of this is thatmore common tokens, like public in Figure 1 are assigned a full subword unit, whereas less common tokens, like setter, are divided into smaller units that are individually more common.

在我们的代码LM中，我们通过让模型在RNN的每个时间步骤预测子字单元而不是完整的标记来解决这个问题。 子词单元是n-gram字符，它们作为语料库中某个标记的子序列出现。 图1显示了分割成子字单元的Java源文件示例。请注意，我们包含一个特殊的子字符单元</ t>，用于标记令牌的结尾，允许我们将一系列子字单元转换回 令牌序列。 模型中的子词单元是根据统计频率自适应地选择的，我们将在稍后描述。 这样做的结果是，更常见的令牌，如图1中的公共，被分配了一个完整的子词单元，而不太常见的令牌，如setter，被分成更单独更常见的较小单元。

The use of a subword unit LM for code has two potential advantages. First, because the model has a smaller vocabulary size, it may have better performance because of a reduced level of data sparsity. Second, the model can synthesize OOV tokens that have not been seen in the training data via the smaller subtoken units. The vocabulary of subword units is learned before training the NLM by segmenting a corpus of code. This is done in such a way that more frequent character n-grams are more likely to be included in the vocabulary of subwords units. This strategy results in a core vocabulary of subword units that occurs frequently across different projects and captures statistical patterns of characters within identifiers.

使用子单词单元LM作为代码有两个潜在的优势。首先，由于模型拥有更小的词汇表大小，因此由于数据稀疏性的降低，它可能具有更好的性能。其次，该模型可以通过较小的子令牌单元合成训练数据中没有出现的OOV令牌。在训练NLM之前，通过对语料库进行分段来学习子单词单元的词汇。这样做的方式是，更频繁的字符n-gram更有可能包含在子单词单元的词汇表中。这种策略产生了一个由子单词单元组成的核心词汇表，这些子单词单元经常跨不同的项目出现，并在标识符中捕获字符的统计模式。

In order to learn the segmentation we use a modification of byte pair encoding (BPE) [31]. BPE is a data compression algorithm that iteratively finds the most frequent pair of bytes in the vocabulary appearing in a given sequence, and then replaces it with a new unused entry. Sennrich, Haddow, and Birch [65] first adapted the algorithm for word segmentation so that instead of merging pairs of bytes, it merges pairs of characters or character sequences. The learned segmentation was used in their neural translation system and resulted in improved translation of rare words.

为了学习分割，我们使用了一个修改的字节对编码(BPE)[31]。BPE是一种数据压缩算法，它迭代地找到词汇表中出现在给定序列中的最频繁的字节对，然后用一个新的未使用的条目替换它。Sennrich、Haddow和Birch[65]首先将该算法应用于分词，这样就不用合并字节对，而是合并字符对或字符序列。将所学习的分割方法应用到他们的神经翻译系统中，提高了罕见词的翻译水平。

The algorithm starts with a vocabulary containing all single characters in the data set plus the </t> symbol. All symbol pairs appearing in the vocabulary are counted and we then replace all the appearances of the most frequent pair (S1, S2) with a unique new single symbol S1S2, which we also add to the vocabulary. This procedure is called a merge operation (S1, S2) ! S1S2. The algorithm stops after a given maximum number of merge operations to be performed is reached. We clarify that as in [65] we do not consider merging pairs that cross token boundaries, that is, where the merged token would contain </t> internally, so that every subword unit is a character subsequence of a token in the data. The final output of the algorithmis the new vocabulary,which contains all the initial characters plus the symbols created from the merge operations, and the ordered list of merge operations performed in each iteration. We run the BPE algorithm on a held out dataset of projects that are separate from the training, validation, and test sets. We experimented with three different encoding sizes, i.e., the maximum number of merge operations: 2000, 5000, and 10000 operations.

该算法从一个词汇表开始，该词汇表包含数据集中所有单个字符加上</t>符号。计算词汇表中出现的所有符号对，然后用一个惟一的新符号S1S2替换最频繁出现的所有符号对(S1, S2)，我们还将该符号添加到词汇表中。这个过程称为合并操作(S1, S2) !S1S2。当要执行的合并操作达到给定的最大数目时，算法停止。我们在[65]中阐明，我们不考虑合并跨令牌边界的对，即合并后的令牌内部包含</t>，因此每个子单词单元都是数据中令牌的字符子序列。算法的最终输出是新词汇表，其中包含所有初始字符和从合并操作创建的符号，以及在每次迭代中执行的合并操作的有序列表。我们在与培训、验证和测试集分离的项目数据集中运行BPE算法。我们实验了三种不同的编码大小，即，合并操作的最大数量:2000、5000和10000操作。

To train the LM, we first segment the train, validation, and test sets using the learned encoding. To do this, we transform each token into a sequence of its characters, adding </t> symbols after every token. Then we apply in order the merge operations from BPE to merge the characters into subword units in the vocabulary. Finally, we train and test a GRU LM in the usual way on the data that has been segmented into subword units.

为了训练LM，我们首先使用学习编码对训练、验证和测试集进行分段。为此，我们将每个令牌转换为其字符序列，在每个令牌之后添加</t>符号。然后，我们按照BPE中的合并操作的顺序将字符合并到词汇表中的子单词单元中。最后，我们用通常的方法对划分为子单词单元的数据进行训练和测试。

3.3 Predicting Best k Tokens

In an autocompletion setting, it might be desirable to present a ranked list of k predicted tokens rather than a single best prediction. But because our model is based on subword units, it is not completely trivial to generate top k predictions of full tokens, because a single token could be made from many subword units. We approximate these using a beam-search-like algorithm. If the beam is large enough the algorithm can give a good approximation of the top-k complete tokens.

在自动完成设置中，最好是显示k个预测令牌的排序列表，而不是单个最佳预测。但是因为我们的模型是基于子单词单位的，所以生成完整标记的前k个预测并不完全是简单的，因为一个标记可以由许多子单词单位组成。我们使用一种类似于波束搜索的算法来近似它们。如果波束足够大，该算法可以很好地逼近前k个完整令牌。

More specifically, the NLM defines a probability p(s1 . . . sN ) for any sequence of subword units. The goal of the search procedure is: given a history s1 . . . sN of subword units that already appear in a source file, predict the complete token that is most likely to occur next. A complete token is a sequence of subword units w1 . . .wM that comprise exactly one token: that is, wM ends with </t> and none of the earlier subword units do. The goal of the search algorithm is to find the k highest probability complete tokens, where we denote a single token as the sequence of units w1 . . .wM, that maximize the model’s probability p(w1 . . .wM |s1 . . . sN ). 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.

更具体地说，NLM定义了一个概率p(s1…对于任何子单词单元序列。搜索过程的目标是:给定一个历史s1…已经出现在源文件中的子单词单元的sN，预测接下来最有可能发生的完整令牌。一个完整的标记是一组子单词单元w1 . .wM的序列，它只包含一个标记:也就是说，wM以</t>结尾，而前面的子单词单元都没有这样做。搜索算法的目标是找到k个概率最大的完整令牌，其中我们将单个令牌表示为单元w1 . . wm的序列，使模型的概率p(w1 . . wm |s1 . .sN)。重要的是，新的完整令牌的长度M不是预先固定的，但是目标是搜索不同长度的完整令牌。

The algorithm is illustrated in Algorithm 1. Given a value of k and a beam size b, it starts by querying the model to obtain its predictions of possible subword units, ranked by probability; in our pseudocode, we assume that the model’s predict function returns a ranked list of a given size, and that V is the total size of the vocabulary. 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. The candidate class has an extend method which updates both of these fields in order to add one additional subword unit to the end of the candidate. Both of the priority queues are sorted by the probability of the candidate.

算法如算法1所示。给定k和a波束大小为b的值，首先查询模型，获得其对可能的子单词单位的预测，按概率排序;在伪代码中，我们假设模型的predict函数返回一个给定大小的排序列表，V是词汇表的总大小。该算法使用两个优先级队列:一个称为candidate，它对搜索过程中仍然需要搜索的子单词单元序列进行排序;另一个称为besttoken，它包含迄今为止已扩展的k个概率最高的完整令牌。每个候选项都是一个包含两个字段的结构，text是候选项中所有子单词单元的连接，prob是候选项中每个子单词单元的概率的乘积。candidate类有一个扩展方法，该方法更新这两个字段，以便在candidate的末尾添加一个额外的子单词单元。这两个优先队列都是根据候选队列的概率排序的。

The main loop of the search is in lines 9-23. In each iteration, the algorithm pops the b best candidates from the candidates queue, expanding them and scoring their expansions, in which each candidate is extended by one additional subword unit. If an expansion creates a token, that is, 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. This search procedure is repeated until any of the following termination criteria has been satisfied at line 9:

(a) The number of complete tokens that have been explored during the search exceeds a threshold (in our implementation, we use tokensDone > 5000).

(b) The cumulative probability of all the tokens that have been explored exceeds the threshold, i.e. total > 0.8

(c) A sufficient number of search iterations have been completed, i.e. iters > 7.

(d) The probability of the best candidate is less than the worst current complete top-k tokens, that is, min{c.prob | c 2 bestTokens} \_ max{c.prob | c 2 candidates)}.

Expanding a candidate cannot increase its probability, so at this point we are guaranteed that no better complete tokens will be found in the remainder of the search. These criteria ensure that the beam search always terminates.

搜索的主循环在第9-23行。在每次迭代中，算法从候选队列中取出b个最佳候选项，对它们进行扩展，并对它们的扩展进行评分，其中每个候选项都被一个额外的子单词单元扩展。如果展开创建了一个令牌，即新的子单词单元以</t>结束，则

它被推到令牌队列中，然后弹出最差的令牌。这维护了besttoken大小为k的不变式。如果新的扩展不是一个完整的令牌，那么它将被推到候选队列中，在下一个迭代中它可能会被扩展。此搜寻程序会重复进行，直至符合下列任何一项终止准则(第9行)为止:

(a)在搜索过程中探索的完整令牌数量超过一个阈值(在我们的实现中，我们使用tokensDone > 5000)。

(b)所有被探索的令牌的累积概率都超过了阈值，即总> 0.8

(c)完成了足够数量的搜索迭代，即iter > 7。

(d)最佳候选的概率小于当前最坏的全top-k令牌，即min{c。prob | c 2 best token} \_ max{c。prob | c 2 candidate)}。

展开候选项不能增加它的概率，因此在这一点上，我们可以保证在搜索的其余部分中不会找到更好的完整令牌。这些条件确保波束搜索总是终止。

3.4 Dynamic adaptation to new projects

It is important to be able to quickly adapt a global LM, which has been trained on a diverse corpus of projects, to have better performance on a new project of interest. We call this dynamic adaptation. For example, suppose that an organization distributes an IDE that contains a code LM trained on a large number of Github projects, and a separate company is using the IDE for a new project. The LM will be expected to perform worse on the new project [40], so for performance it would be desirable to retrain the model on the new project. However, for confidentiality reasons, the company may well be unwilling to send their code back to the IDE vendor for retraining the model. Although a within-project model might be an alternative, a single project does not provide much training data for an NLM, especially in the early stages of a project. Therefore, it would be desirable to have a fast way of adapting the LM to a new project, in a way that requires access only to the trained LM and the source code of the new project. In principle, we could train from scratch a new model on both the original training set and the new project, but this would be computationally expensive.

重要的是能够快速适应已经在多样化项目集中训练的全球LM，以在新的感兴趣项目上获得更好的表现。我们称之为动态适应。例如，假设组织分发包含在大量Github项目上训练的代码LM的IDE，并且一个单独的公司正在将IDE用于新项目。预计LM将在新项目中表现更差[40]，因此对于性能，需要在新项目上重新训练模型。但是，出于保密原因，公司可能不愿意将其代码发送回IDE供应商以重新培训模型。虽然项目内部模型可能是一种替代方案，但单个项目不能为NLM提供太多的培训数据，尤其是在项目的早期阶段。因此，希望有一种快速的方式使LM适应新项目，其方式只需要访问训练好的LM和新项目的源代码。原则上，我们可以从头开始训练原始训练集和新项目的新模型，但这在计算上是昂贵的。

Instead, we use a simple method of dynamically adapting our global neural LMs to a new project. Given a new project, we start with the global LM 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. Our choice of applying only one update is motivated by the following 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 LM to give too much weight to the new project, losing information about the large training set of the global model.

相反，我们使用一种简单的方法来动态调整我们的全局神经LM到一个新项目。 给定一个新项目，我们从全局LM开始，并通过在测试之后对项目中每个遇到的序列采取单个梯度步骤来更新模型参数。 这一系列更新相当于新项目的单一培训时代。 （在我们第6节的评估中，我们将以一种我们从未在我们的测试集上进行培训的方式拆分项目文件。）我们将GRU展开20个时间步而不是200个，就像我们的全局模型一样，以便 更频繁地更新参数。 我们选择仅应用一次更新的原因如下。 首先，它更快，允许模型快速适应项目中的新标识符。 其次，在新项目上采取过多的梯度步骤可能会导致LM对新项目给予过多的重视，从而失去有关全球模型的大型训练集的信息。

4 DATASETS

In our experiments we used code corpora from three popular programming languages: Java, C, and Python. Although these languages are related, they also have differences that might be hypothesized to affect the performance of LMs. Java was an obvious choice since it has extensively been used in related work [6, 22, 37, 40, 58, 69]. Unlike Java, C is not object oriented, and the language makes it possible to write exceptionally terse code.3 Finally, Python is also object oriented but it is mainly a dynamic language with little use of static typing. These differences between C and Python make them interesting to consider alongside Java.

在我们的实验中，我们使用了三种流行编程语言的代码库:Java、C和Python。虽然这些语言是相关的，但是它们也有可能影响LMs性能的差异。Java是一个显而易见的选择，因为它在相关工作中得到了广泛的应用[6、22、37、40、58、69]。与Java不同的是，C语言不是面向对象的，这种语言使得编写异常简洁的代码成为可能。最后，Python也是面向对象的，但它主要是一种动态语言，很少使用静态类型。C语言和Python之间的这些差异使它们与Java一起值得考虑。

For Java we used the Java Github corpus of Allamanis et al. [6], which consists of more than 14000 popular open source Java projects. Following the procedure described in [6], the C corpus was mined in [26] and the Python corpus was mined in [28]. For lexical analysis in Java we used the lexer implemented in [37]4, while C and Python code lexical analysis was performed via the Pygments library. In Python, we do not add any special tokens to represent whitespace. For all three languages, we preprocessed the data by replacing occurrences of non-ASCII character sequences such as Chinese ideograms inside strings with a special token that did not occur elsewhere in the corpus.

对于Java，我们使用了Allamanis等人的Java Github语料库。按照[6]中描述的过程，在[26]中挖掘C语料库，在[28]中挖掘Python语料库。对于Java中的词法分析，我们使用了在[37]4中实现的lexer，而C和Python代码的词法分析是通过Pygments库执行的。在Python中，我们不添加任何特殊的令牌来表示空格。对于所有这三种语言，我们都对数据进行了预处理，将字符串中出现的非ascii字符序列(如中文表意文字)替换为语料库中没有出现的特殊标记。

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 upon which BPE was run to learn a subword encoding. 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 [37].We divide the data into five subsets as in the other two languages. The validation and test sets are the same as in [37], 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.

对于Python和C，我们抽取1%的语料库进行验证，1%的语料库进行测试。另外10%的语料库作为一个单独的数据集进行采样，在此基础上运行BPE以学习子单词编码。其余数据用于培训。我们还报告了整个训练集2%的一小部分的结果。对于Java，我们使用了一个稍微不同的过程，使我们的实验可以与之前的研究[37]进行比较。与其他两种语言一样，我们将数据分成五个子集。验证和测试集[37]是一样的,我们的“小火车”是一样的训练集,获得完整的Java训练集,我们收集的所有文件在Java不发生在Github语料库验证或测试集,这些我们取样1000个随机项目subword编码数据集,剩下的项目作为完整的训练集。

5 EVALUATION

Our model was evaluated on both intrinsic and extrinsic evaluation measures. An intrinsic metric judges the quality of an LM’s predictions by themselves, in isolation from a larger task. On the other hand, extrinsic methods perform indirect evaluation by assessing how a model affects performance of some other task. We next describe the specific metrics used in our experiments.

我们的模型在内在和外在评估指标上进行了评估。 内在指标独立于较大的任务，自行判断LM预测的质量。 另一方面，外在方法通过评估模型如何影响某些其他任务的性能来执行间接评估。 我们接下来将介绍我们实验中使用的具体指标。

5.1 Intrinsic evaluation

A good language model should assign high probability to a real sentence while simultaneously assigning a low probability to a wrong one. In many applications in software engineering, accurate probability scoring is necessary, meaning that 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 like translating a program from one programming language to another [45, 56], code completion [29, 64], and code synthesis from natural language and vice versa [7, 14, 23, 25, 57, 61].

一个好的语言模型应该为一个真实的句子分配高概率，同时为一个错误的句子分配低概率。在软件工程的许多应用程序中，准确的概率评分是必要的，这意味着更可能出现在人类编写的代码中的代码片段应该被分配更高的概率。对于将程序从一种编程语言转换成另一种编程语言、代码完成[29,64]以及从自然语言转换成代码合成[7,14,23,25,57,61]等任务来说，对代码片段进行精确的评分是必不可少的。

The intrinsic metric that we use is cross entropy, which is a standard measure employed in previous work. The cross entropy defines a score over a a sequence of code tokens t1, t2, ..., t |C | . For each token ti , the probability of each token is estimated using the model under evaluation and it is denoted by p(ti |t1, ..., ti−1). Then the average per token entropy is defined as:

我们使用的内在度量是交叉熵，这是以前工作中使用的标准度量。 交叉熵定义了一系列代码令牌t1，t2，...，t | C |的分数。 对于每个标记ti，使用所评估的模型估计每个标记的概率，并且用p（ti | t1，...，ti-1）表示。 然后每个标记熵的平均值定义为：

Cross entropy corresponds to the average number of bits required in every prediction. Thus lower values are better. This metric not only takes into account whether the highest ranked prediction is correct, but also rewards predictions with high confidence.

交叉熵对应于每个预测中所需的平均比特数。 因此，较低的值更好。 该指标不仅考虑排名最高的预测是否正确，还考虑高可信度的奖励预测。

Our subword unit models define a distribution over subword units rather than directly over tokens. To compute the cross entropy for subword unit models, we segment each token ti into subword units ti = wi1 . . .wiM. Then we compute the product p where the right hand side can be computed directly by the subword unit NLM. This probability allows us to compute the cross entropy Hp (C). (The technical reason that this method is correct is that discrete probability distributions are preserved under 1:1 correspondences.)

我们的子词单元模型定义了子词单元的分布，而不是直接在令牌上。 为了计算子字单元模型的交叉熵，我们将每个标记ti分割成子字单元ti = wi1。。.WIM。 然后我们计算产品p，其中右侧可以由子字单元NLM直接计算。 这个概率允许我们计算交叉熵Hp（C）。 （该方法正确的技术原因是离散概率分布以1：1的对应关系保存。）

5.2 Extrinsic evaluation

As an extrinsic performance measure, we report the performance of our LMs on code completion, which is the task of predicting each token in a test corpus given all of the previous tokens in the file. To measure performance on this task, we use mean reciprocal rank (MRR). MRR has previously been used in a plethora of code completion evaluations in relevant work [13, 37, 64, 69]. The reciprocal rank of a query response is the multiplicative inverse of the rank of the first correct answer. MRR is the average of reciprocal ranks or results for a sample of queries Q defined as

作为外在性能度量，我们报告了LM在代码完成方面的性能，这是在给定文件中所有先前令牌的情况下预测测试语料库中的每个令牌的任务。 为了衡量这项任务的绩效，我们使用平均互惠等级（MRR）。 MRR以前曾在相关工作中用于过多的代码完成评估[13,37,64,69]。 查询响应的倒数等级是第一个正确答案的等级的乘法逆。 MRR是定义为Q的查询样本的倒数排名或结果的平均值

For example, if the correct suggestion always occurs at rank 2 then the MRR is 0.5 roughly, at rank 10 the MRR is 0.1, and so on. A simplified description of MRR is that it averages top-k prediction accuracy across various k. In this specific scenario k 2 [1, 10] since the models output a list of top-10 best tokens.

例如，如果正确的建议总是出现在等级 2，那么 MRR 大致是 0.5，在等级 10，MRR 是 0.1，依此类推。MRR 的一个简化描述是，它平均了各种 k 的 top-k 预测精度。在这个特定的场景中，k 2 [1,10] 因为模型输出了前 10 名最佳令牌的列表。

5.3 Test Scenarios

Our model was evaluated in three scenarios introduced in previous work [37], which they call static, dynamic, and maintenance settings. Each setting simulates a different way of incorporating code LMs within an IDE. For all settings, the task is to predict each token in the test set, but the training sets for each setting are slightly different.

我们的模型是在前面的工作[37]中引入的三个场景中评估的，它们分别称为静态、动态和维护设置。每种设置都模拟了在IDE中集成代码LMs的不同方式。对于所有设置，任务是预测测试集中的每个令牌，但是每个设置的训练集略有不同。

Static tests.

The model is first trained on a fixed training corpus, and is later evaluated on a separate test dataset. Essentially this is a cross-project setting where the train, validation, and tests are all disjoint from each other and contain separate projects. This simulates the setting where a single global LM is trained on a large corpus of projects and then deployed to many different customers without any adaption.

该模型首先在固定的训练语料库上进行训练，然后在单独的测试数据集上进行评估。本质上，这是一个跨项目设置，其中培训、验证和测试都彼此不相交，并包含单独的项目。这模拟了一种设置，即在大量项目上训练单个全局 LM，然后部署到许多不同的客户，而无需任何调整。

Dynamic tests.

In this setting, the model is allowed to update its parameters after it has made predictions on files in the test set. In addition to the original training set, the model is allowed to retrain on files in the test set, after it has been scored on its predictions. Note that the model is required to make its predictions on the testing file before the file is added to the training set, so that we are never training on test data. For our neural LMs, we adapt the model at test time using the procedure described in Section 3.4. After we have finished evaluating each test project we restore the model to the global cross-project one learned from the train set. This simulates a setting in which some files are available from the test project of interest for dynamic adaptation.

在此设置中，允许模型在对测试集中的文件进行预测后更新其参数。 除了原始训练集之外，在对其预测进行评分之后，允许模型重新训练测试集中的文件。 请注意，在将文件添加到训练集之前，模型需要对测试文件进行预测，因此我们从不对测试数据进行过培训。 对于我们的神经LM，我们使用3.4节中描述的程序在测试时调整模型。 在我们完成每个测试项目的评估之后，我们将模型恢复到从列车集中学到的全局跨项目。 这模拟了一个设置，其中一些文件可从感兴趣的测试项目中获得，以进行动态调整。

Software Maintenance tests.

This scenario is perhaps the closest to real world usage. It simulates everyday development where a programmer makes small changes to existing code. In this setting, the LMs are tested on one file at a time in the test set. For each file, the full training set plus all other files in the test project apart from the file of interest is used as training data. Because this requires retraining the model once for each file in the test set, this scenario was previously deemed infeasible for NLMs in [37].

这种情况可能是最接近现实世界的用法。 它模拟了程序员对现有代码进行细微更改的日常开发。 在此设置中，LM在测试集中一次在一个文件上进行测试。 对于每个文件，除了感兴趣的文件之外，完整的训练集加上测试项目中的所有其他文件被用作训练数据。 因为这需要为测试集中的每个文件重新训练模型一次，所以此方案以前被认为是[37]中的NLM不可行的。

6 RESEARCH QUESTIONS

When evaluating our models, we focus on the following research questions.

在评估我们的模型时，我们关注以下研究问题。

RQ1. How does the performance of subunit neural LMs compare to state-of-the-art LMs for code?

In this RQ, we evaluate whether the use of subword units allows the training of effective neural language models. Specifically, we compare subword unit NLMs to standard n-gram language models [40], cache LMs [69], state-of-the-art n-gram models with nested caching [37], and token-level NLMs [74]. Unfortunately, it was not possible to PHOG [12] because the public implementation of PHOG does not include the code for the first stage that generates the HOG rules. This stage is necessary to be able to run the implementation on any dataset other than the one used in the original paper [12]. Their dataset is both smaller than our full dataset, and the split is based on a within project scenario, rather than the cross-project scenario that is the focus of our paper, so it is not possible to address our RQs using their data. We also do not compare to the recent pointer network augmented RNN [48] as it was evaluated on the dataset of [12] and no implementation of their model is available.

RQ1。子单元神经LM的性能与最先进的LM代码相比如何？

在这个RQ中，我们评估子字单元的使用是否允许训练有效的神经语言模型。具体来说，我们将子词单元NLM与标准n-gram语言模型[40]，缓存LMs [69]，具有嵌套缓存的最先进的n-gram模型[37]和令牌级NLM [74]进行比较。 。不幸的是，PHOG [12]是不可能的，因为PHOG的公共实现不包括生成HOG规则的第一阶段的代码。这个阶段对于能够在原始论文[12]中使用的数据集之外的任何数据集上运行实现是必要的。他们的数据集都小于我们的完整数据集，并且拆分基于项目内部场景，而不是我们论文重点关注的跨项目场景，因此无法使用他们的数据来解决我们的RQ问题。我们也没有比较最近的指针网络增强RNN [48]，因为它是在[12]的数据集上进行评估，并且没有可用的模型实现。

As described in Section 3, we hypothesize that NLMs might outperform all of the n-gram models because of their ability to take larger context into account, and that subword unit NLMs might perform better than token level NLMs because of the improved ability to handle OOV and rare words. We do not report results for character-level models since as discussed in Section 2 these models have not been proved to offer improvement in NLP and their use is impractical. We also do not include results for subtoken models segmented via the heuristic in [3] as in preliminary experiments, we found that subtoken models were less effective than token level models, so we chose to not include them in this comparison. Following the previous work, we evaluate the models on the Github Java dataset (Section 4) using the evaluation framework described in Section 6.

如第3节所述，我们假设NLMs可能优于所有n-gram模型，因为它们能够考虑更大的上下文，而子单词单元NLMs可能比标记级NLMs执行得更好，因为它改进了处理OOV和罕见单词的能力。我们不报告字符级模型的结果，因为正如第2节所讨论的，这些模型还没有被证明能够改进NLP，并且它们的使用是不切实际的。在[3]中，我们也没有将启发式分割的子令牌模型的结果包含在实验中，我们发现子令牌模型不如令牌级模型有效，所以我们选择不将它们包含在这次的比较中。在前面的工作之后，我们使用第6节中描述的评估框架评估Github Java数据集(第4节)上的模型。

RQ2. Are subword unit NLMs effectively trainable on large code corpora, such as giga-token corpora?

Does the additional training data yield significant performance improvements? Training on a larger corpus can usually be expected to improve themodel’s performance, but this is not guaranteed, because the impact of more data tends to have diminishing returns, as after some point the model’s performance saturates and does not continue to improve with more data. This saturation point will be different for different models, so it is an interesting research question to ask whether neural LMs can make better use of large corpora than n-gram models, because NLMs are more complex models that can take into account larger amounts of context.

RQ2。 子字单元NLM是否可以在大型代码语料库上有效训练，例如千兆令牌语料库？

额外的培训数据是否会显着提高性能？ 通常可以预期对更大的语料库进行培训可以提高模型的性能，但这并不能保证，因为更多数据的影响往往会带来收益递减，因为在某些时候模型的性能饱和并且不会随着更多数据而持续改善。 对于不同的模型，这个饱和点将是不同的，因此询问神经LM是否可以比n-gram模型更好地利用大型语料库是一个有趣的研究问题，因为NLM是更复杂的模型，可以考虑更大量的上下文。

However, training on larger data uses more resources because the amounts of parameters that have to be estimated also grows with the amount of training data. For example, n-gram language models need to estimate counts for every new n-gram that appears in the training corpus, while NLMs need to learn input and output embedding matrices whose dimension scales with the vocabulary size. Token-level models are especially difficult to train on large corpora since they have particular difficulty with the huge vocabularies. A huge vocabulary results in a massive model that is unable to fit in any modern GPU and would be too slow to train even with the help of softmax approximation techniques such as noise contrastive estimation [36] and adaptive softmax [33]. Furthermore, the use of approximation techniques could potentially cause a small decrease in the model’s performance. Character level models will be too slow to train on very large training sets and impractical to use. This problem is also especially relevant to code LMs, because the vocabulary size for a code corpus grows much more quickly than a natural language corpus. For example, the one billion words benchmark corpus for English has a vocabulary of 0.8 million words [16], while our similar size training set for Java has a vocabulary of 6.5 million when in both cases all words are discarded with count below 3. This huge difference in vocabulary size is also true for other programming languages. Specifically, the training set used in our experiments for Java, C, and Python have vocabulary sizes 10.5, 8, and 13 million respectively when no vocabulary threshold is used. Additionally, while OOV rate is only 0.32% for the English corpus test set, for the Java one, it is larger than 13%. The model required for a token-level code corpus of this size cannot fit in modern GPUs and is only trainable on multiple CPU threads using parallelization and softmax approximation techniques[16].

但是，对较大数据的培训使用了更多资源，因为必须估计的参数数量也随着训练数据量的增加而增加。例如，n-gram语言模型需要估计出现在训练语料库中的每个新n-gram的计数，而NLM需要学习其维度与词汇量大小一致的输入和输出嵌入矩阵。令牌级模型特别难以在大型语料库上训练，因为它们对于庞大的词汇表特别困难。巨大的词汇量导致了一个无法适应任何现代GPU的大规模模型，即使借助于诸如噪声对比度估计[36]和自适应softmax [33]之类的softmax近似技术，也无法进行训练。此外，使用近似技术可能会导致模型性能的小幅下降。角色等级模型太慢，无法在非常大的训练集上进行训练，并且使用起来不切实际。这个问题也与代码LM特别相关，因为代码语料库的词汇量大小比自然语言语料库快得多。例如，十亿字的英语基准语料库的词汇量为80万字[16]，而我们类似规模的Java训练集的词汇量为650万，而在这两种情况下，所有单词都被丢弃，数量低于3。词汇量的巨大差异对其他编程语言也是如此。具体而言，我们的Java，C和Python实验中使用的训练集在没有使用词汇阈值时分别具有10.5,8和1300万的词汇量。此外，虽然英语语料库测试集的OOV率仅为0.32％，但对于Java语言测试集，其大于13％。这种大小的令牌级代码语料库所需的模型不适合现代GPU，只能使用并行化和softmax近似技术在多个CPU线程上进行训练[16]。

RQ3. How does the performance of subword unit NLMs vary across programming languages?

In principle the learning methods forNLMs are language agnostic; however, the majority of studies evaluate only on Java code, so it is an important research question to verify that current LM techniques for code are equally effective on other programming languages. One might hypothesize that the terseness of C or the lack of static type information in Python would make it more difficult to achieve good performance from an LM. We test this hypothesis by measuring cross entropy and MRR in the static and dynamic setting across the three language corpora described in Section 4.

RQ3。 子字体单元NLM的性能如何因编程语言而异？

原则上，NLM的学习方法与语言无关; 然而，大多数研究只评估Java代码，因此验证当前用于代码的LM技术在其他编程语言中是否同样有效是一个重要的研究问题。 有人可能会假设C的简洁性或Python中缺少静态类型信息会使得从LM获得良好性能变得更加困难。 我们通过测量第4节中描述的三种语言语料库中的静态和动态设置中的交叉熵和MRR来测试该假设。

RQ4. Is the dynamic updating procedure effective at dynamically updating subword unit NLMs to new projects?

Past research has focused on the strong locality that characterizes code [40, 62, 69]. As a consequence, we expect new projects to introduce many new identifiers that do not appear even in a large cross-project corpus. For this reason, it has been shown that n-gram models can benefit significantly from dynamically adapting to the test corpus, as described in Section 6. In this RQ we ask whether NLMs can also benefit from dynamic adaptation, and whether the procedure that we introduce in Section 3.4 is effective at dynamically adapting NLMs to new projects. We test this hypothesis by comparing our dynamic adaption method for subword unit NLMs against two advanced n-gram models that have been shown to benefit from adaptation: cache LMs [69] and nested cache LMs [37]. We use the dynamic and software maintenance settings described in Section 6, following [37]. A naive approach to the software maintenance setting would require retraining the model from scratch for every file in the test corpus, which was rightly deemed to be infeasible for NLMs by [37]. Instead, we apply our dynamic adaptation procedure from Section 3.4, which is much more efficient because it trains for only one epoch on each test file.

RQ4。动态更新过程在动态更新子单词单元NLMs到新项目上有效吗?

过去的研究集中于代码的强局部性[40,62,69]。因此，我们希望新项目引入许多新的标识符，这些标识符甚至不会出现在大型跨项目语料库中。由于这个原因，n-gram模型可以从对测试语料库的动态适应中获得显著的好处，如第6节所述。在这个RQ中，我们询问NLMs是否也可以从动态适应中获益，以及我们在第3.4节中介绍的过程在动态适应NLMs以适应新项目方面是否有效。我们通过将子单词单元NLMs的动态适应方法与两个已被证明受益于自适应的高级n-gram模型(缓存LMs[69]和嵌套缓存LMs[37])进行比较来验证这一假设。我们使用[37]之后的第6节中描述的动态和软件维护设置。一种简单的软件维护设置方法需要对测试语料库中的每个文件从头开始重新训练模型，[37]正确地认为这对于NLMs是不可行的。相反，我们应用了第3.4节中的动态适应过程，它的效率更高，因为它只在每个测试文件上训练一个epoch。

7 RESULTS

7.1 RQ1. Performance of models

In Tables 2 and 3, we show the performance of the different models on the static, dynamic, and maintenance settings for Java. Note that for cross entropy, lower numbers are better, whereas for MRR, higher numbers are better. The entropy results for the n-gram models are copied from [37]; these are comparable to ours because we use the same training and test split as their work. For MRR evaluation we used v0.1 of their model hosted in GitHub, which is the version used in their reported experiments.6 In [37] the reported MRR had been calculated on the entirety of the test set). While for their NLM baselines it was measured only on the first 1 million tokens of the test set. For this reason we recalculated MRR for the above on the first 1 million tokens of our test set. The closed vocabulary NLM is our own implementation.

在表2和表3中，我们展示了Java的静态，动态和维护设置的不同模型的性能。 注意，对于交叉熵，较低的数字更好，而对于MRR，较高的数字更好。 n-gram模型的熵结果复制自[37]; 这些与我们的相似，因为我们使用与他们的工作相同的培训和测试分组。 对于MRR评估，我们使用了在GitHub中托管的模型的v0.1，这是他们报告的实验中使用的版本。在[37]中，报告的MRR已经在整个测试集上计算得出。 虽然对于他们的NLM基线，它仅在测试集的前100万个令牌上进行测量。 出于这个原因，我们在测试集的前100万个令牌中重新计算了上述MRR。 闭合词汇NLM是我们自己的实现。

From the tables it can be seen that on both metrics, our open vocabulary NLM has better predictive performance than any of the n-gram models, even the nested cache models of [37] that are designed specifically for code. To specifically evaluate the effect of relaxing the closed vocabulary assumption, we compare our open vocabulary NLM to a closed vocabulary one. The closed vocabulary NLM uses exactly the same architecture as our open vocabulary models (single layer GRU with the same model size and hyperparameters), but is trained on complete tokens rather than subword units. We note that the closed vocabulary NLM that we report has much better results than the NLM language model that is used as a baseline in [37]; this is primarily because our model incorporates a fully connected hidden layer, and also dropout, which has been shown to improve the performance of RNN LMs [50]. So our token-level NLM baseline is much harder to beat than those reported in previous work. Even so, we find that our open vocabulary NLM has much better predictive performance than the closed vocabulary model. The difference in performance between the open and closed vocabulary NLMs is larger for the dynamic and maintenance settings than for the static setting. We hypothesize that this is because in the open vocabulary model, dynamic adaptation can help the model to learn patterns about OOV words in the test set; this is not possible for a model with a closed vocabulary.

从表中可以看出，在两个指标上，我们的开放词汇表NLM具有比任何n-gram模型更好的预测性能，甚至是[37]专为代码设计的嵌套缓存模型。为了具体评估放松闭合词汇假设的效果，我们将我们的开放词汇NLM与封闭词汇表进行比较。闭合词汇表NLM使用与我们的开放词汇表模型（具有相同模型大小和超参数的单层GRU）完全相同的架构，但是在完整的令牌而不是子词单元上进行训练。我们注意到，我们报告的封闭词汇表NLM比在[37]中用作基线的NLM语言模型具有更好的结果;这主要是因为我们的模型包含一个完全连接的隐藏层，还有丢失，这已被证明可以改善RNN LMs的性能[50]。因此，我们的令牌级NLM基线比以前的工作中报告的要难得多。即便如此，我们发现我们的开放词汇NLM比闭合词汇模型具有更好的预测性能。开放和闭合词汇NLM之间的性能差异对于动态和维护设置而言比静态设置更大。我们假设这是因为在开放词汇模型中，动态适应可以帮助模型学习测试集中OOV词的模式;对于具有封闭词汇表的模型，这是不可能的。

We report the performance of the open vocabulary NLMs with different vocabulary sizes, obtained after 2000, 5000, and 10000 BPE merge operations. We see that performance is similar across the different vocabulary sizes, indicating that a large vocabulary size is not required for good performance.

我们报告了不同词汇量的开放词汇NLMs在2000、5000和10000个BPE合并操作后的性能。我们看到不同词汇表大小之间的性能是相似的，这表明良好的性能并不需要很大的词汇表大小。

Finally, following [37], note that we cannot report results for the nested or cache n-gram models on the static setting because these models make use of information from the test. Consequently, even if we do not adapt their global model on the test set, their additional components are always adapted on it. Also, following [37] we do not report results from the closed vocabulary NLM on the maintenance setting due to the massive time and space requirements of this experiment.

最后，在[37]之后，请注意我们无法在静态设置上报告嵌套或缓存n-gram模型的结果，因为这些模型使用了测试中的信息。 因此，即使我们不在测试集上调整他们的全局模型，他们的附加组件也总是适应它。 此外，在[37]之后，由于本实验的大量时间和空间要求，我们不会报告关闭词汇表NLM对维护设置的结果。

Based on these results, we conclude that even when trained on a relatively small corpus, open vocabulary NLMs are effective models of code. Indeed, to our knowledge, our model has state of the art performance on this data set.

基于这些结果，我们得出结论，即使在相对较小的语料库上进行训练，开放词汇表NLM也是有效的代码模型。 实际上，据我们所知，我们的模型在该数据集上具有最先进的性能。

7.2 RQ2. Large Corpora

When trained on larger corpora the performance of traditional n-gram models and their variations like the nested cache model gets saturated and they are unable to effectively leverage the extra information [37]. In contrast, our model is able to better leverage the increase in training data as shown in Tables 2 and 3. As expected the entropy of our NLM decreased significantly, by about 1.5 bits and MRR increased by about 6% for all encoding sizes in the static scenario when trained on the full corpus. Essentially, this means that the additional training data helps our NLM learn to synthesize identifiers from subword units better and with higher confidence.

当对较大的语料库进行训练时，传统的n-gram模型及其变体(如嵌套缓存模型)的性能趋于饱和，无法有效地利用额外的信息[37]。相比之下，我们的模型能够更好地利用表2和表3所示的培训数据的增加。正如预期的那样，我们的NLM的熵显著降低，在静态场景中，当对全部语料库进行训练时，所有编码大小的NLM的熵都降低了约1.5位，MRR增加了约6%。本质上，这意味着额外的训练数据帮助我们的NLM更好地、更有信心地从子单词单元合成标识符。

The improvements are smaller but still exist when the model is dynamically adapted on a test project. For all encoding sizes the models improve by 0.5 bits in entropy and by about 2 to 3% in MRR. In contrast, the nested cache n-gram model entropy decreases by less than 0.1 bits and MRR less than 0.4%. Similar improvements for the nested cache n-gram model were also reported in [37], supporting our findings. From that we conclude that subword unit NLMs can utilize a large code corpus better than n-gram models. However, if one lacks a model trained on large corpus or there is not enough time to train one, then satisfactory performance can still be achieved by training on a small corpus.

当模型在测试项目上动态调整时，改进较小但仍然存在。 对于所有编码大小，模型在熵方面提高0.5位，在MRR方面提高约2至3％。 相反，嵌套高速缓存n-gram模型熵减小小于0.1位且MRR小于0.4％。 [37]也报道了嵌套缓存n-gram模型的类似改进，支持了我们的研究结果。 由此我们得出结论，子字单元NLM可以比n-gram模型更好地利用大代码语料库。 然而，如果缺少一个在大型语料库中训练的模型或者没有足够的时间训练一个模型，那么通过训练小语料库仍然可以实现令人满意的表现。

In addition, we note that training on giga-token or larger corpora is scalable. Table 4 shows the VRAM requirements for training when a BPE segmentation with 2000, 5000, 10000 operations and a batch size of 32 is used. Obviously, training time will be a lot larger than the smaller set, but we see from the earlier results that the model’s predictive performance of the open vocabulary models was stable across different vocabulary sizes. This means that we can use a relatively moderate vocabulary size with the open vocabulary NLM and still obtain good performance. As training is a one-off process that does not need to be repeated and a pretrained model can be downloaded and loaded in a GPU in a matter of seconds, this results in a real time applicable model even when trained on huge corpora. More importantly, as reported by [37] a token level NLM with a vocabulary of only 76K tokens required a few days to be trained on the smaller training corpus. Training our model on the same data was a matter of less than 12 hours on a single GPU.

此外，我们注意到giga-token或更大的语料库的培训是可扩展的。表4显示了使用2000,5000,10000操作和批量大小为32的BPE分段时的VRAM培训要求。显然，训练时间将比较小的训练时间大得多，但我们从早期的结果中看出，开放词汇模型的模型预测性能在不同的词汇量中是稳定的。这意味着我们可以使用开放词汇表NLM使用相对适中的词汇量，并且仍然可以获得良好的性能。由于训练是一次性过程，不需要重复，并且可以在几秒钟内下载并加载到GPU中的预训练模型，即使在大型语料库上训练，这也会产生实时适用的模型。更重要的是，正如[37]报道的那样，只有76K代币词汇表的令牌级NLM需要在较小的训练语料库上训练几天。在相同的数据上训练我们的模型在单个GPU上的时间不到12小时。

Lastly, our model can be integrated in IDEs to facilitate code completion as queries for the next token can be answered in real time and the required memory is less than that of training since batching is no longer necessary. The decreased memory requirements are illustrated in Table 5 and are less than 400MBs for any of the encodings we used.

最后，我们的模型可以集成到IDE中以便于代码完成，因为可以实时回答对下一个令牌的查询，并且由于不再需要批处理，所需的内存小于训练的内存。 表5中说明了降低的内存要求，对于我们使用的任何编码，都小于400MB。

7.3 RQ3. Multiple Languages

In Tables 6 and 7 we report the performance of our open vocabulary NLMs on Java, C, and Python. Tables 8 and 9 present the results for the dynamic setting. We see that performance on C and Python is at least as good as Java, providing evidence that our methodology for training subword unit NLMs is indeed language agnostic. We caution the reader to not interpret these results as a comparison of the programming languages as to which is more predictable, which is more terse, etc. The first reason for this caution is that the training corpora have slightly different sizes across the different languages. Unfortunately, it does not seem possible to define a fair notion of "same training set size" across programming languages, because tokens in one language might be more informative than others, e.g. Python code has a larger proportion of identifiers. Even if it were possible to do this, different languages have different standard libraries and are typically used to solve problems in different domains. All of these concerns pose serious threats to validity to any attempt to compare programming languages via language modeling, so we do not attempt to draw such conclusions in this work.

在表6和表7中，我们报告了Java，C和Python上的开放词汇表NLM的性能。表8和表9给出了动态设置的结果。我们看到C和Python上的性能至少与Java一样好，这表明我们用于训练子词单元NLM的方法确实与语言无关。我们提醒读者不要将这些结果解释为编程语言的比较，因为它更可预测，更简洁，等等。这种谨慎的第一个原因是训练语料库在不同语言中的大小略有不同。不幸的是，似乎不可能在编程语言中定义“相同训练集大小”的公平概念，因为一种语言中的令牌可能比其他语言更具信息性，例如： Python代码具有更大比例的标识符。即使可以这样做，不同的语言也有不同的标准库，通常用于解决不同域中的问题。所有这些问题都对通过语言建模比较编程语言的任何尝试的有效性构成严重威胁，因此我们不会尝试在这项工作中得出这样的结论。

7.4 RQ4. Dynamic Adaptation

Finally, to evaluate the effect on the dynamic adaptation method for our subword unit NLMs, consider again the results in Tables 2 and 3. As [37] point out, it is straightforward to adapt an n-gram LM, because we can simply add and remove counts. Indeed, we see that all of the advanced n-gram models in the dynamic and maintenance settings perform better than any of the NLM models in the static setting. This result holds both for the small train set and for the full train set. In other words, the improvement due to dynamic adaptation is greater than the improvement due to an NLM. Once we apply the dynamic adaptation method to our Open Vocabulary NLM, however, then the picture changes. With dynamic adaptation, our model achieves better cross-entropy than the current state-of-the-art [37]. From this we conclude that our dynamic adaptation method is indeed effective at fine-tuning a global subword unit NLM to a specific test project.

最后，为了评估对我们的子单元NLM的动态自适应方法的影响，再次考虑表2和3中的结果。如[37]所指出的，可以直接调整n-gram LM，因为我们可以简单地添加 并删除计数。 实际上，我们发现动态和维护设置中的所有高级n-gram模型都比静态设置中的任何NLM模型表现更好。 该结果适用于小火车组和整列火车组。 换句话说，由动态适应引起的改进大于由NLM引起的改进。 然而，一旦我们将动态自适应方法应用于我们的开放词汇表NLM，那么图像就会改变。 通过动态适应，我们的模型实现了比现有技术更好的交叉熵[37]。 由此我们得出结论，我们的动态自适应方法确实可以有效地将全局子字单元NLM微调到特定的测试项目。

We note that evaluating NLMs on this scenario was previously deemed infeasible since multiple models had to be created each trained on the entirety of the test set minus one file. Nevertheless, the small size of our model allowed the experiments for this scenario to be completed in a few days. We achieved this by training each model only on information from the same project. For large test projects, we first split them into multiple partitions and for each one we trained a model on the rest. All files from the same partition can then load this model and need to only train on other files from the same partition. This strategy offered considerable speed gains. Furthermore, the experiment could be sped up significantly as parallelization is fairly easy and both memory and computation requirements are fairly small, thus achievable even with a single GPU.

我们注意到，在这个场景中评估NLMs之前被认为是不可行的，因为必须创建多个模型，每个模型都要经过整个测试集减去一个文件的训练。尽管如此，由于我们的模型规模较小，因此这个场景的实验可以在几天内完成。我们通过仅根据来自相同项目的信息对每个模型进行培训来实现这一点。对于大型测试项目，我们首先将它们划分为多个分区，并为每个分区训练一个模型。然后，来自相同分区的所有文件都可以加载这个模型，并且只需要对来自相同分区的其他文件进行培训。这种策略提供了相当大的速度增益。此外，由于并行化相当容易，而且内存和计算需求都比较小，因此即使使用一个GPU，也可以显著加快实验速度。

8 CONCLUSIONS

We have presented a new open-vocabulary neural language model for source code. By defining the model on subword units, which are character subsequences of tokens, the model is able to handle neologisms, that is, new identifier names which have not appeared in its training data, while keeping the size of the model relatively small. We are able to train a neural language model on over one billion tokens of code, a data set over a hundred times larger than had been used for previous neural LMs for code. On the problem of predicting the next token, the resulting model outperforms recent state-of-the-art models based on adding nested caches to n-gram language models. We hope that the simplicity of our model will allow advances in deep learning for code by allowing the implementation of more complex architectural ideas such as attention [8, 71]. Also, improved language models for code have the potential to enable new tools for aiding code readability [2], program repair [11, 15, 35, 62], program synthesis [34] and translation between programming languages [45, 56]. Finally, the general technique of using subword units is not limited to language modeling, but can easily be incorporated into many neural models of code tokens. Therefore, we hope that this idea could have broad application throughout software engineering, such as in models to suggest readable function and class names [3], summarizing source code [5, 43], predicting bugs [60], detecting code clones [73], comment generation [42], fixing syntactic errors [47], and variable deobfuscation [9].

我们为源代码提出了一种新的开放式词汇神经语言模型。通过在作为标记的字符子序列的子字单元上定义模型，该模型能够处理新词，即，未在其训练数据中出现的新标识符名称，同时保持模型的大小相对较小。我们能够训练超过十亿个代码令牌的神经语言模型，这个数据集比以前用于代码的神经LM的数据集大一百倍。在预测下一个令牌的问题上，得到的模型优于基于向n-gram语言模型添加嵌套高速缓存的最新技术。我们希望通过允许实现更复杂的架构思想（例如注意力[8,71]），我们模型的简单性将允许深入学习代码。此外，改进的代码语言模型有可能启用新工具来帮助代码可读性[2]，程序修复[11,15,35,62]，程序综合[34]和编程语言之间的转换[45,56]。最后，使用子字单元的一般技术不仅限于语言建模，而是可以很容易地结合到代码令牌的许多神经模型中。因此，我们希望这个想法可以在整个软件工程中得到广泛应用，例如在模型中建议可读函数和类名[3]，总结源代码[5,43]，预测错误[60]，检测代码克隆[73] ]，评论生成[42]，修复句法错误[47]和变量反混淆[9]。