Are Deep Neural Networks the Best Choice for Modeling Source Code?

ABSTRACT

Current statistical language modeling techniques, including deep learning based models, have proven to be quite effective for source code. We argue here that the special properties of source code can be exploited for further improvements. In this work, we enhance established language modeling approaches to handle the special challenges of modeling source code, such as: frequent changes, larger, changing vocabularies, deeply nested scopes, etc. We present a fast, nested language modeling toolkit specifically designed for software, with the ability to add & remove text, and mix & swap out many models. Specifically, we improve upon prior cache-modeling work and present a model with a much more expansive, multi-level notion of locality that we show to be well-suited for modeling software. We present results on varying corpora in comparison with traditional N-gram, as well as RNN, and LSTM deep-learning language models, and release all our source code for public use. Our evaluations suggest that carefully adapting N-gram models for source code can yield performance that surpasses even RNN and LSTM based deep-learning models.

1 INTRODUCTION

There has been much interest in the idea of “naturalness": viz., modeling and exploiting the repetitive nature of software using statistical techniques from natural language processing (NLP) [17, 26, 38].Statistical models from NLP, estimated over the large volumes of code available in GitHub, have led to a wide range of applications in software engineering. High-performance language models are widely used to improve performance on NLP-related tasks, such as translation, speech-recognition, and query completion; similarly, better language models for source code are known to improve performance in tasks such as code completion [15]. Developing models that can address (and exploit) the special properties of source code is central to this enterprise.

Language models for NLP have been developed over decades, and are highly refined; however, many of the design decisions baked-into modern NLP language models are finely-wrought to exploit properties of natural language corpora. These properties aren’t always relevant to source code, so that adapting NLP models to the special features of source code can be helpful. We discuss 3 important issues and their modeling implications in detail below.

**Unlimited Vocabulary**

Code and NL can both have an unbounded vocabulary; however, in NL corpora, the vocabulary usually saturates quickly: when scanning through a large NL corpus, pretty soon, one rarely encounters new words. New proper nouns (people & place names) do pop up–but do so infrequently. Code is different; while each language only has a fixed set of keywords and operators, new identifier names tend to proliferate [4]. Modeling Implications: In NLP, it’s de regeur to limit vocabulary to the most common e.g., 50,000 words in a pre-processing step, before model estimation. Words outside this vocabulary are treated as an unknown word, or omitted entirely. This artificially limits the space of events over which to distribute probability mass. Similarly, numerals and strings are replaced with generic tokens. This works for NLP, since words outside the dominant vocabulary are so rare. Virtually all work in modeling of source code borrows this approach. In source code, given the constant vocabulary innovation, this approach is not appropriate. We demonstrate that a closed vocabulary (even if large) does indeed negatively affect performance (Section 5.4), and introduce methods to address this.

无限词汇量代码和NL都可以有无限词汇量;然而，在NL语料库中，词汇量通常很快就会饱和:当扫描一个大型的NL语料库时，很快就很少会遇到新单词。新的专有名词(人名和地名)确实会出现，但并不常见。代码是不同的;虽然每种语言只有一组固定的关键字和操作符，但是新的标识符名称往往会大量增加[4]。建模含义:在NLP中，将词汇表限制在最常见的范围内是必要的，例如，在模型估计之前的预处理步骤中有50,000个单词。在这个词汇表之外的单词被视为一个未知的单词，或者被完全省略。这人为地限制了事件的空间，使其无法分布概率质量。类似地，数字和字符串被通用标记替换。这对NLP是有效的，因为主导词汇表之外的单词非常罕见。实际上，源代码建模中的所有工作都借用了这种方法。在源代码中，考虑到词汇表的不断创新，这种方法是不合适的。我们演示了封闭词汇表(即使很大)确实会对性能产生负面影响(第5.4节)，并介绍了解决这个问题的方法。

**Nested, Scoped, Locality**

While developers do invent new names for variables, classes and methods, the repeated use of these names tends to be localized. In Java, e.g., local variables, parameters and private methods can be introduced & used, repeatedly, in one scope, and never used elsewhere. The package structures in large systems can introduce nesting of such vocabulary scopes, with different identifiers going in and out of use as one traverses the package hierarchy [7, 24, 34, 36]. Researchers have even noted application and developer-specific vocabularies [35]. Modeling Implications: This type of nested, scoped vocabulary innovation is accompanied by corresponding repetition, where certain code structures involving specific local names repeat, locally, within their own nested scopes. This requires a nested modeling approach, which captures the local repetition within a scope si , and then makes it available to scopes si ,si+1 . . . nested within si . Furthermore, if such nested models are used within an interactive tool (such as an IDE) the model would need to be rapidly re-estimated as the programmer’s working context changes.

虽然开发人员确实为变量、类和方法创建了新的名称，但是这些名称的重复使用往往是本地化的。例如，在Java中，局部变量、参数和私有方法可以在一个范围内重复引入和使用，而在其他地方则不能使用。大型系统中的包结构可以引入此类词汇表范围的嵌套，在遍历包层次结构时使用不同的标识符。研究人员甚至注意到了应用程序和特定于开发人员的词汇[35]。建模含义:这种类型的嵌套、作用域词汇表创新伴随着相应的重复，其中涉及特定本地名称的特定代码结构在它们自己的嵌套作用域内本地重复。这需要一种嵌套的建模方法，它捕获范围si内的本地重复，然后使其对范围si,si+1可用…嵌套在si中。此外，如果在交互式工具(如IDE)中使用这种嵌套模型，则需要随着程序员的工作上下文的更改快速重新评估模型。

**Dynamism**

Evolution is normal for well-used software systems; bug fixes and new features keep rolling in. NLP corpora evolve much more slowly. Furthermore, during interactive coding, software tools must quickly adjust to new localities and contexts: in a single coding session, a developer may open and close many files. As she explores the code, a language model that works within the IDE (for code completion [15, 33], defect localization [31], etc.) must rapidly adapt to the working context. Modeling Implications: Traditional NLP models cannot handle rapid re-estimation. Deep-learning models, in particular, are not very dynamic, and re-estimation is very slow.

In response to the observations and concerns raised above, we have developed a dynamic, hierarchically scoped, open vocabulary language model for source code, that achieves best-in-class performance when using non-parametric (count-based) language modeling. We make the following contributions:

• We introduce mixed, scoped models to handle arbitrary nesting and mixing of N-gram models.

• We implement these models using a fast data structure optimized for dynamic, scoped counting of language events.

• We compare several popular smoothing techniques in related work and show that a simple approach (not typically used) works better than others.

• Finally, we evaluate the performance of these models on a large corpus of Java code in comparison and combination with implicit (deep-learning based) models. We find that our model outperforms the RNN and LSTM deep learning models, achieving unprecedented levels of entropy & also performance on the code-suggestion task. We also show that our approach adds value, even to LSTM models. Our runnable API, code and replication details can be found on github.com/SLP-Team/SLP-Core

对于使用良好的软件系统，进化是正常的;错误修复和新功能不断涌现.NLP语料库的发展速度要慢得多。此外，在交互式编码期间，软件工具必须快速适应新的地点和环境：在单个编码会话中，开发人员可以打开和关闭许多文件。在探索代码时，在IDE中工作的语言模型（代码完成[15,33]，缺陷本地化[31]等）必须快速适应工作环境。建模含义：传统的NLP模型无法处理快速重估。特别是深度学习模型不是非常动态，重新估计非常慢。

针对上面提出的观察和关注，我们为源代码开发了一个动态的，分层范围的，开放的词汇表语言模型，在使用非参数（基于计数的）语言建模时实现了最佳的性能。我们做出以下贡献：

•我们引入混合的范围模型来处理N-gram模型的任意嵌套和混合。

•我们使用针对语言事件的动态范围计数优化的快速数据结构来实现这些模型。

•我们在相关工作中比较了几种流行的平滑技术，并表明一种简单的方法（通常不使用）比其他方法效果更好。

•最后，我们通过与隐式（基于深度学习的）模型进行比较和组合，评估这些模型在大型Java代码库中的性能。我们发现我们的模型优于RNN和LSTM深度学习模型，在代码建议任务上实现了前所未有的熵水平和性能。我们还表明，我们的方法甚至为LSTM模型增加了价值。我们的可运行API，代码和复制细节可以在github.com/SLP-Team/SLP-Core上找到

7 CONCLUSION

We have made the following contributions.

• We introduce a dynamically updatable, nested scope, unlimited vocabulary count-based N-gram model that significantly outperforms all existing token-level models, including very powerful ones based on deep learning. Our model is far faster than the deep learning models. Our nested cache model achieves an MRR performance of 0.818, with unlimited vocabulary (0.85 with limited vocabulary) which is best-in-class.. Our work illustrates that traditional approaches, with some careful engineering, can beat deep learning models.

• We show that our count-based approach “plays well" with LSTM models, and yields even better performance in combination, particularly in terms of entropy scores where the best mixture achieving 1.25 bits of entropy per token without constraining the vocabulary.

• Our detailed evaluations reveal some new observations:

(1) Jelinek-Mercer smoothing outperforms smoothing approaches used in prior work.

(2) Limiting vocabularies artificially and misleadingly boosts intrinsic performance, without boosting actual performance on the suggestion task.