A Survey of Machine Learning for Big Code and Naturalness

Research at the intersection of machine learning, programming languages, and software engineering has recently taken important steps in proposing learnable probabilistic models of source code that exploit code’s abundance of patterns. In this article, we survey this work. We contrast programming languages against natural languages and discuss how these similarities and differences drive the design of probabilistic models. We present a taxonomy based on the underlying design principles of each model and use it to navigate the literature. Then, we review how researchers have adapted these models to application areas and discuss crosscutting and application-specific challenges and opportunities.

机器学习，编程语言和软件工程的交叉研究最近采取了重要步骤，提出了可利用代码丰富模式的可学习的源代码概率模型。 在本文中，我们调查了这项工作。 我们将编程语言与自然语言进行对比，并讨论这些相似之处和差异如何推动概率模型的设计。 我们提出了基于每个模型的基本设计原则的分类法，并用它来导航文献。 然后，我们将回顾研究人员如何将这些模型应用于应用领域，并讨论横切和特定应用的挑战和机遇。

1 INTRODUCTION

Software is ubiquitous in modern society. Almost every aspect of life, including healthcare, energy, transportation, public safety, and even entertainment, depends on the reliable operation of high-quality software. Unfortunately, developing software is a costly process: software engineers need to tackle the inherent complexity of software while avoiding bugs, and still delivering highly functional software products on time. There is therefore an ongoing demand for innovations in software tools that help make software more reliable and maintainable. New methods are constantly sought, to reduce the complexity of software and help engineers construct better software.

软件在现代社会中无处不在。 生活的几乎每个方面，包括医疗保健，能源，交通，公共安全，甚至娱乐，都取决于高质量软件的可靠运行。 不幸的是，开发软件是一个代价高昂的过程：软件工程师需要解决软件固有的复杂性，同时避免错误，并且仍能按时交付功能强大的软件产品。 因此，不断需要软件工具的创新，以帮助使软件更可靠和可维护。 不断寻求新方法，以降低软件的复杂性并帮助工程师构建更好的软件。

Research in this area has been dominated by the formal, or logico-deductive, approach. Practitioners of this approach hold that, since software is constructed in mathematically well-defined programming languages, software tools can be conceived in purely formal terms. The design of software tools is to be approached using formal methods of definition, abstraction, and deduction. Properties of tools thus built should be proven using rigorous proof techniques such as induction over discrete structures. This logico-deductive approach has tremendous appeal in programming languages research, as it holds the promise of proving facts and properties of the program. Many elegant and powerful abstractions, definitions, algorithms, and proof techniques have been developed, which have led to important practical tools for program verification, bug finding, and refactoring [24, 42, 45]. It should be emphasized that these are theory-first approaches. Software constructions are viewed primarily as mathematical objects, and when evaluating software tools built using this approach, the elegance and rigor of definitions, abstractions, and formal proofs-of properties are of dominant concern. The actual varieties of use of software constructs, in practice, become relevant later, in case studies, that typically accompany presentations in this line of work.

这一领域的研究一直由正式或逻辑演绎方法主导。这种方法的实践者认为，由于软件是用数学上定义良好的编程语言构建的，因此可以用纯粹的形式术语来构思软件工具。使用正式的定义，抽象和演绎方法来处理软件工具的设计。这样构建的工具的属性应该使用严格的证明技术来证明，例如对离散结构的感应。这种逻辑演绎方法在编程语言研究中具有巨大的吸引力，因为它有望证明程序的事实和属性。已经开发了许多优雅而强大的抽象，定义，算法和证明技术，这些都为程序验证，错误查找和重构提供了重要的实用工具[24,42,45]。应该强调的是，这些是理论优先的方法。软件构造主要被视为数学对象，在评估使用此方法构建的软件工具时，属性的定义，抽象和形式证明的优雅和严谨性是主要关注点。在实践中，软件构造的实际使用种类后来在案例研究中变得相关，这通常伴随着这一系列工作中的陈述。

Of late, another valuable resource has arisen: the large and growing body of successful, widely used, open-source software systems. Open-source software systems such as Linux, MySQL, Django, Ant, and OpenEJB have become ubiquitous. These systems publicly expose not just source code, but also meta-data concerning authorship, bug-fixes, and review processes. The scale of available data is massive: billions of tokens of code and millions of instances of meta-data, such as changes, bug-fixes, and code reviews (“big code”). The availability of “big code” suggests a new, data-driven approach to developing software tools: why not let the statistical distributional properties, estimated over large and representative software corpora, also influence the design of development tools? Thus rather than performing well in the worst case, or in case studies, our tools can perform well in most cases, thus delivering greater advantages in expectation. The appeal of this approach echoes that of earlier work in computer architecture: Amdahl’s law [15], for example, tells us to focus on the common case. This motivates a similar hope for development tools, that tools for software development and program analysis can be improved by focusing on the common cases using a fine-grained estimate of the statistical distribution of code. Essentially, the hope is that analyzing the text of thousands of well-written software projects can uncover patterns that partially characterize software that is reliable, easy to read, and easy to maintain.

最近，出现了另一种宝贵的资源：成功的，广泛使用的开源软件系统。开源软件系统，如Linux，MySQL，Django，Ant和OpenEJB已经无处不在。这些系统不仅公开了源代码，还公开了有关作者，错误修复和审核流程的元数据。可用数据的规模巨大：数十亿代码令牌和数百万个元数据实例，例如更改，错误修复和代码审查（“大代码”）。 “大代码”的可用性提出了一种新的，数据驱动的软件工具开发方法：为什么不让大型和有代表性的软件语料库中的统计分布属性也影响开发工具的设计？因此，在最坏的情况下或在案例研究中，我们的工具不是表现良好，而是在大多数情况下表现良好，从而在期望方面提供更大的优势。这种方法的吸引力与早期计算机体系结构的工作相呼应：例如，Amdahl定律[15]告诉我们要关注常见案例。这激发了对开发工具的类似希望，通过使用对代码统计分布的细粒度估计来关注常见情况，可以改进软件开发和程序分析的工具。从本质上讲，希望分析数千个编写良好的软件项目的文本可以发现部分表征可靠，易读且易于维护的软件的模式。