生成式预训练语言模型在代码管理上的性能分析

刘一尘[1]

[1](南京邮电大学 计算机学院、软件学院、网络空间安全学院,江苏 南京 210000)
通讯作者: 刘一尘

摘 要: PLM 近年来受到了广泛关注,这些基于 Transformer 结构的预训练语言模型已经彻底改变了 NLP 领域。一般来说,这些模型可以分为三类:基于编码器结构的掩码语言模型、基于解码器结构的自回归语言模型、编码器-解码器模型。随着生成式预训练语言模型在项目管理,包括代码生成、日志监测等领域的初步应用,形如 Github Copilot、AixCoder 等生成式预训练语言模型已经能够在代码编辑辅助与日志管理管理等环节帮助用户减轻负担。其中,生成式预训练语言模型在执行代码生成任务时,仍然存在非独立函数生成效果劣于独立函数生成效果的问题;在执行日志管理任务时,存在模型不能理解特定于领域的术语、难以充分捕获完整的日志上下文信息、难以获得同一日志的不同样式的通用表示等问题,导致这类生成式预训练语言模型难以在实际应用环境中推广使用。通过针对生成式预训练语言模型开发更具泛用性的评估数据集,可以对不同代码生成模型的能力进行量化,促进多类型生成式预训练语言模型之间的合作;通过对这类模型在日志管理的过程中进行性能调优,可以显著提高生成式语言模型在拥有相关领域知识时对日志的理解能力。

关键词: 生成式预训练语言模型;代码生成;日志管理

Performance Analysis of Generative Pre-trained Language Models in Code Management

Liu Yichen[1]

[1](School of Computer Science, Nanjing University of Posts and Telecommunications, Nanjing 210000, China)

Abstract: In recent years, PLM (Pre-trained Language Model) has garnered significant attention, and these pre-trained language models based on the Transformer architecture have revolutionized the NLP (Natural Language Processing) field. Generally speaking, these models can be categorized into three types: encoder-based masked language models, decoder-based autoregressive language models, and encoder-decoder models. With the initial application of generative pre-trained language models in project management, including code generation, log monitoring, and other areas, generative pre-trained language models such as Github Copilot and AixCoder have been able to assist users in reducing their workload in code editing assistance and log management.However, when it comes to code generation tasks, generative pre-trained language models still face issues such as the inferior performance in generating non-independent functions compared to independent functions. In the context of log management, these models struggle with understanding domain-specific terminology, capturing complete log context information, and obtaining a general representation of different styles of the same log. These challenges hinder the widespread adoption of generative pre-trained language models in practical application environments.By developing more generalized evaluation datasets tailored for generative pre-trained language models, we can quantify the capabilities of different code generation models and promote collaboration between various types of generative pre-trained language models. Additionally, performance optimization during the log management process of these models can significantly improve their understanding of logs when they possess relevant domain knowledge.

Key words: generative pre-trained language models; code generation; log monitoring

生成式预训练语言模型近年来在代码生成、日志管理环节出现了飞跃式的进展,Github Copilot 等代码辅助编辑模型降低了代码编写的门槛,并为现有的程序编辑人员提供了较大便利,以至于许多现有的项目代码中都存在由生成式预训练语言模型所产生的痕迹,这类模型生成代码的基础直接影响到软件开发的质量,作为软件开发者,我们总是将目光关注在这类模型所生成的代码能否正确运行,而忽视了生成代码的质量,以及预训练语言模型对于代码日志反馈调优的分析能力,这种尚不成熟的开发捷径影响了软件开发与维护所产生的潜在成本。因此,针对生成式预训练语言模型在代码质量分析量化与日志理解调优相关领域的研究一直

是软件测试领域的一大重要话题。本文整理了三篇与生成式预训练语言模型测试相关的论文，通过对这类论文进行阅读，除了对现有的生成式预训练语言模型在代码生成、日志管理领域的进展获取大致的了解之外，也能够学习到量化生成代码的质量与日志分析管理的常用方法与实际应用技巧。

1 知识增强的日志理解预训练语言模型

论文《KnowLog: Knowledge Enhanced Pre-trained Language Model for Log Understanding》[1]发布于 ICSE 2024，其介绍了一种知识增强的日志理解预训练语言模型 KnowLog，设计了三个新的预训练任务来有效地利用领域知识来提高对日志的理解，分别为缩写预测任务、描述判别任务以及日志对比学习任务。有效地利用领域知识来提高预训练语言模型对日志的理解。

1.1 研究问题

这篇文章的主要研究对象是生成式预训练语言模型在日志理解领域的实际情况。作者提出了知识增强的日志理解预训练语言模型 KnowLog，它提高了日志理解任务的最先进性能，并提出了三个新的预训练任务，有效地利用领域知识来增强模型效果，使日志的表示更加通用。针对模型不能理解特定于领域的术语，尤其是缩写的问题，提出了缩写预测任务，利用缩写信息作为局部知识，使模型能够准确理解这些领域缩写；针对模型难以充分捕获完整的日志上下文信息的问题，提出了一个描述判别任务，通过利用文档中的自然语言描述知识作为全局知识来补充日志的底层知识并充分捕获完整的日志上下文信息；针对模型难以获得同一日志的不同样式的通用表示的问题，提出了一个日志对比学习任务，以捕获日志中的共性，并使日志的表示更加通用。在知识增强的预训练之后，KnowLog 在六个不同的下游任务上进行了微调。与其他没有知识增强的预训练模型相比，KnowLog 达到了最先进的性能。

1.2 相关工作

许多研究提出了使用预训练语言模型进行日志理解。这些工作可以分为两类。一类工作是不对预训练模型进行微调。这些预训练模型的参数不会被更新，而是仅获取日志的表示向量，并将其输入到其他深度模型中来完成特定任务。已有的 HitAnomaly[2]、SwissLog[3]、NeuralLog[4]和 ClusterLog[5]利用预训练模型进行基于日志的异常检测；另一类工作是对预训练模型进行微调。通过在一个小任务数据集上对整个预训练模型进行微调，可以获得一个特定任务的模型。NuLog[6]、Logstamp[7]和 LogPPT[8]被提出用于对预训练模型进行微调以实现日志解析。同样，Translog[9]和 LogEncoder[10]也基于日志异常检测进行了微调。

然而，现有的方法仍然存在三个弱点：首先，这些模型不能理解特定于领域的术语，尤其是缩写；其次，这些模型难以充分捕获完整的日志上下文信息；第三，这些模型难以获得同一日志的不同样式的通用表示。

UniLog 首先提出在日志上进行预训练，目标是重建掩码令牌，然后对多个下游任务进行微调。本文认为，UniLog 的简单预训练任务和缺乏领域知识不能解决上文提出的挑战。

基于这些工作，可以发现没有深入的研究解决日志理解的挑战。本文提出了利用领域知识加强日志预训练的新视角，并设计了相关的预训练任务，以提高日志的理解能力。

1.3 研究方法

为了解决上述问题，作者提出了知识增强的日志理解预训练语言模型 KnowLog，该模型遵循一种预训练和微调范式，在具有知识增强的日志数据上进行预训练，并在不同的下游任务上进行微调。具体来说，作者使用日志作为预训练语料库，并将日志的缩写和自然语言描述分别作为局部知识和全局知识，设计了三个新的预训练任务来有效地利用领域知识来提高对日志的理解。具体来说：针对模型不能理解特定于领域的术语，尤其是缩写的问题，提出了缩写预测任务，利用缩写信息作为局部知识，使模型能够准确理解这些领域缩写；针对模型难以充分捕获完整的日志上下文信息的问题，提出了一个描述判别任务，通过利用文档中的自然语言描述知识作为全局知识来补充日志的底层知识并充分捕获完整的日志上下文信息；针对模型难以获得同一日志的不同样式的通用表示的问题，提出了一个日志对比学习任务，以捕获日志中的共性，并使日志的表示更加通用。

针对输入表示，日志中的令牌不仅包含一些自然语言的单词，还包含一些领域缩写，以及一些专家根据自己的习惯构造的符号，这些现象对如何有效地分割日志提出了挑战。为了避免词汇外(OOV)问题并有效地保留标记信息，本文使用预训练语言模型的标记器 tokenizer 进行分词。对于给定的输入文本，作者使用标记

器来获得令牌序列作为编码器的输入。作为一个特殊的标记，并且其最后的隐藏表示被用来整合整个序列的表示。

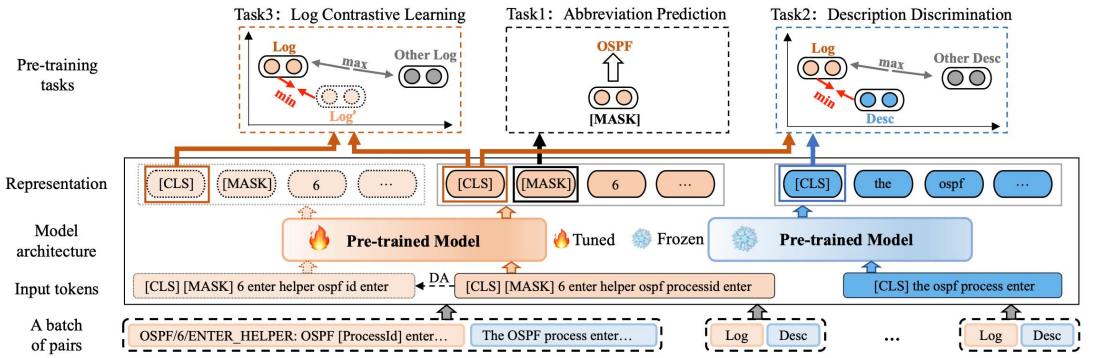
为了通过相应的自然语言描述增强对日志的理解，本文构建了“日志－描述对”并采用双编码器架构，其中两个编码器分别处理每种类型的数据。本文使用基于 Transformer[11]仅编码器结构的预训练模型作为主干对日志进行编码，给定一个标记化后的标记序列，编码器输出该序列的隐藏表示。

在模型的预训练环节中，该方法首先屏蔽日志中的缩写，并对日志执行数据增强。然后，两个预训练的语言模型分别对日志和描述进行编码，得到相应的表示。最后，将这些表征用于三个新的预训练任务。第一个任务是缩写预测，它利用缩写作为本地知识来理解这些缩写。第二个任务是描述判别，它利用日志的自然语言描述作为全局知识来补充日志的底层知识，并使模型能够充分捕获完整的日志上下文信息。第三个任务是日志对比学习，它基于对比学习来捕获日志中的共性，并获得更通用的日志表示。

缩写预测任务的两个问题在于：目前的模型无法有效理解缩略语、当使用标记器处理日志时，这些缩写被分割成子词标记，这破坏了缩写的语义。为了有效地捕获日志中领域术语的语义，文中利用缩写信息作为局部知识来增强模型。首先从多个供应商的公共文档中的术语表中收集了 1711 个缩写。其次，为了保持缩略词的完整性，将所有缩略词添加到标记器的词汇表中，以确保缩略词不会被切片。最后提出了一种新的预训练缩写预测任务，以增强模型对缩写词的有效理解。

针对描述识别任务，由于日志语法非常简洁，很难有效地捕获日志的底层信息。为了更好地理解日志，文中利用日志的自然语言描述作为全局知识来增强模型。该模型能够拉近日志与语义空间中相应的自然语言描述之间的距离，从而减少对日志语义的误解。

在日志对比学习任务中，日志作为由模板和变量组成的半结构化文本，一方面在传递相同语义的日志模板上存在一些差异，另一方面，具有具体参数的日志实例的语义本质上类似于日志模板。



1.4 评估

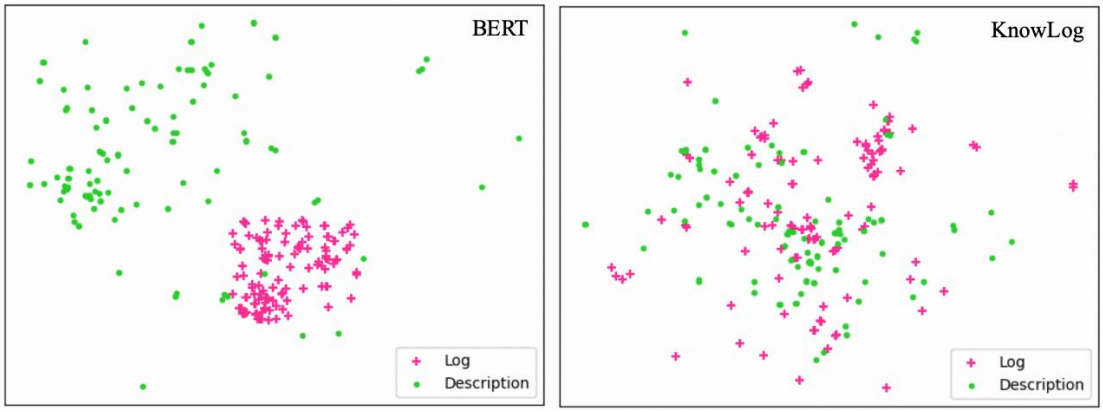
为了评估 KnowLog 处理这些日志的能力，作者构建了六个不同的日志理解下游任务。根据输入的类型，这些下游任务可以分为两类:日志单任务(其中输入是单个日志)和日志对任务(其中输入是日志对或日志对)。对于每个数据集，按 6:2:2 的比例划分训练集、验证集和测试集。在这之中，用训练集对模型进行微调，在验证集中获得最优结果，最后在测试集中评估并报告结果。

任务	作用	数据集	指标
模块分类	用于识别日志所属的模块。该任务的输入是一条模块名被屏蔽的日志，输出是对应的模块名。	使用日志模板作为原始数据，日志中的模块名称作为ground truth，然后将模块名称替换为[MASK]	Accuracy和Weighted F1-score
风险日志识别	以确定日志是否对系统有影响为目标，一种二分类任务	从文档中解析“处理步骤”这个字段，并根据它是否包含明确的操作步骤来构造基本事实。具体来说，带有操作步骤的日志为True，没有操作步骤的日志为False。	Precision、Recall和F1-score
故障现象识别	识别日志所属的故障现象，该任务基于实际数据。	从实际场景中收集了602个华为交换机的日志实例作为数据集，这些日志经过专家标注，共涵盖43种故障现象。	Hamming-score
日志和描述的语义匹配	二分类问题，用于度量日志的语义l是否与相应的自然语言描述相匹配d	从文档中获得日志的描述，然后将日志与其对应的描述标记为True，并随机选择另一个日志标记为False的描述，其中True与False的比例为1:1。	Accuracy and Weighed F1-score
日志和可能原因排序	是一种日志对类型的排序任务，用于根据给定日志的语义上下文确定其最可能的潜在原因。	文档中的每条日志都包含对可能原因的介绍，我们将此字段解析为基础事实。将ground truth和随机选择的5个可能原因的其他日志作为候选集，并要求模型对候选集进行排序。	Precision@1和平均倒数秩(MRR)，Precision@1表示第一名的精度
厂商间模块匹配	一种日志对类型的二进制任务，用于跨供应商对齐相同的模块，它可以度量日志的语义表示是否更通用。	根据模块名称是否相同来构建不同厂商日志之间的ground truth，并随机抽取不同模块的日志作为负样本，其中True与False的比值为1:1。并且将我们将日志中的模块名称替换为[MASK]令牌，以避免训练过程中的标签泄漏。	Accuracyand and Weighed F1-score

为了直观地验证 KnowLog 的有效性，本文选取了四个具有代表性的案例进行定性分析。具体来说，本文采用未经微调的预训练模型来获得输入的嵌入，然后计算嵌入之间的余弦相似度作为度量。选择两种类型的输入进行分析:Log-NL 和 Log-Log。在 Log-NL 类型中，输入由日志和自然语言描述组成。在 Log-Log 类型中，一个输入由两条日志组成。正样本包括一个日志实例及其对应的日志模板，而负样本包括两个不同的日志模板。

Label	Examples	Models	Score
Match	**Log**: BGP/4/ASPATH_EXCEED_MAXNUM: The number of AS-PATHs exceeded the limit([limit-value]). (Operation=[STRING]) **NL**: The number of AS-Paths exceeded the maximum value.	BERT UniLog KnowLog	0.7250 0.7061 0.8006
UnMatch	**Log**: BGP/4/ASPATH_EXCEED_MAXNUM: The number of AS-PATHs exceeded the limit([limit-value]). (Operation=[STRING]) **NL**: The OSPF process successfully exited from GR.	BERT UniLog KnowLog	0.5715 0.3008 0.0056
Match	**Log1**: DEVM/3/hwRemoteFaultAlarm_active(l): The remote fault alarm has occurred. (IfIndex=27, IfName=10GE1/0/17) **Log2**: DEVM/3/hwRemoteFaultAlarm_active: The remote fault alarm has occurred.	BERT UniLog KnowLog	0.9550 0.8031 0.9750
UnMatch	**Log1**: BGP/4/ASPATH_EXCEED_MAXNUM: The number of AS-PATHs exceeded the limit([limit-value]). (Operation=[STRING]) **Log2**: DEVM/3/hwRemoteFaultAlarm_active: The remote fault alarm has occurred.	BERT UniLog KnowLog	0.8338 0.2997 0.1514

KnowLog 能够更好地工作有两个原因。首先，KnowLog 对日志进行了预先训练，并对不同的任务进行了微调。其次，与现有的预训练模型相比，KnowLog 分别利用局部和全局知识来增强预训练。从可视化结果中，我们可以发现基于 BERT[12]的日志表示和相应的自然语言描述在向量空间上是明显分离的，两者几乎是不对齐的。理想情况下，日志应该能够与语义空间中的相应描述保持一致。这表明当前预训练的模型在日志与其对应的描述之间存在明显的语义差距。相比之下，KnowLog 能够将日志和相应的描述拉得更近，使模型能够全面理解日志。



1.5 小结

本文提出了知识增强的日志理解预训练语言模型 KnowLog，它提高了日志理解任务的最先进性能。并且提出了三个新的预训练任务，分别为缩写预测任务、描述判别任务以及日志对比学习任务，有效地利用领域知识来增强模型，使日志的表示更具泛用性。在知识增强的预训练之后，KnowLog 在六个不同的下游任务上进行了微调。与其他没有知识增强的预训练模型相比，KnowLog 达到了最先进的性能，这证明了知识对提高日志理解的有效性。消融分析证明了利用知识来理解日志的这些预训练任务的有效性。

2 使用生成预训练模型生成实用代码的测试

论文《CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models》[13]发布于 ICSE 2024，其提出了一种针对实用代码进行测试分析的新基准，命名为 CodeEval。弥补了评估模型在独立函数上的有效性并不能反映这些模型在实用代码生成场景（即在开源或专有代码的实际设置中生成代码）上的有效性的问题，为实际应用场景中的函数检测提供了全新的测试标准。

2.1 研究问题

近年来，出现了利用机器学习技术，特别是基于 Transformer 的大型生成式预训练语言模型进行代码生成辅助的趋势，如 Codex[14]、AlphaCode[15]、InCoder[16]、CodeGen[17]、PanGu-Coder[18]和 ChatGPT[19]来处理开放领域的代码生成任务。用户通过给定描述所需功能功能的自然语言描述，利用这些模型生成独立函数（即仅调用或访问内置函数和标准库的函数）以及非独立函数。

为了对现有的生成式预训练模型所生成的代码质量进行量化评估，业界广泛选择了使用代码评估测试数据集进行代码质量评估。然而，尽管这些基准测试具有重要性和实用性，但它们与实际的代码生成场景（即针对开源或专有代码的真实设置进行代码生成）之间存在差距。一方面，这些基准测试主要关注独立函数。另一方面，非独立函数在实际的代码生成场景中普遍存在。通过对 GitHub 上分别用 Java 和 Python 编写的 100 个最流行项目进行分析，本文发现非独立函数占开源项目中超过 70%的函数。

为了弥补上述问题，本文作者提出了 CoderEval，一个上下文感知的基准测试，它可以用于评估在实用代码生成方面的代码生成模型。CoderEval 可以为在实用代码生成任务中代码生成模型的有效性提供更实际和具有代表性的评估。该方法是迄今为止唯一支持项目级代码生成并使用生成正确代码能力（Pass@k[14]）评估指标的基准，它可以验证生成代码的功能正确性。同时，作者通过在该测试模型上对现有生成式预训练语言模型依照在大型语言模型生成正确上下文信息的能力（Acc@k[14]）与生成正确代码的能力（Pass@k）标准进行定量分析测试，得出了有建设性的结论。

2.2 相关工作

随着 Codex 的发布，HumanEval[14]成为了一个用于评估代码生成模型生成的 Python 程序功能正确性的基准测试。HumanEval 包含 164 个手写问题，每个问题包括一个函数签名、一个文档字符串、一个标准的参考

函数以及多个单元测试。最近，DS-1000[20]被提出用于评估代码生成模型在生成依赖第三方数据科学库的代码方面的有效性。除了 Python，还有针对 Java（AixBench[21]）和其他编程语言（MultiPL-E[22]）的基准测试，这有助于理解和开发针对这些其他语言的代码生成模型。

2.3 研究方法

作者从流行的实际开源项目中挑选了 230 个 Python 和 230 个 Java 代码生成任务，并创建一个自足的执行平台用于自动评估生成代码的功能正确性。目前，CoderEval 支持六级上下文依赖的代码生成任务，其中上下文指的是在生成函数之外，但在依赖之内的第三方库、当前类、文件或项目中定义的类型、API、变量和常量等代码元素。

CoderEval 的构建包括三个步骤：数据集收集、数据集探索和评估过程。

数据集收集分为：目标选择和构建测试样例。

为了实现目标选择，作者首先通过抓取 GitHub 上所有项目的标签来选择候选项目，并挑选出 14 个标签中出现频率最高且星级较高的项目。对于每个标签，选择星级数最高的前五个项目；随后，提取所选项目中的所有函数，只保留不属于测试、接口或已废弃函数的函数，这些函数要有函数级的英文注释，并能在验证平台中成功运行，且能通过原始测试用例；其次，通过人工筛选从候选函数中选出高质量函数，筛选的主要标准是该函数是否经常出现在实际开发场景中。最后，根据每个项目中包含的被选函数的数量来确定项目。这一过程可以帮助完善后期在编译较少的项目时，选择相同数量的功能。

为了构建合适的测试样例，对于行覆盖率或分支覆盖率未达到 100%的测试，第一作者在 CoderEval 中手动编写了额外的测试用例，以实现函数更高的行覆盖率和分支覆盖率。

在选择函数时，作者确保包含少于 10 个上下文标记的函数。过多的上下文依赖会使模型难以生成正确的解决方案。对于所有大型语言模型来说都难以生成正确实现的函数是不适合用于基准测试的。作者要求 13 名开发人员评判的在实际开发场景中经常使用的函数。这一规则是由于不同的开发人员对常用函数有不同的偏好，CoderEval 中开发人员偏好函数的多样性可以消除潜在的偏差。在选中的函数中，需要包含能反映函数实现的文档说明的函数。实现代码超过三行的函数，并且不属于测试或废弃函数的函数。

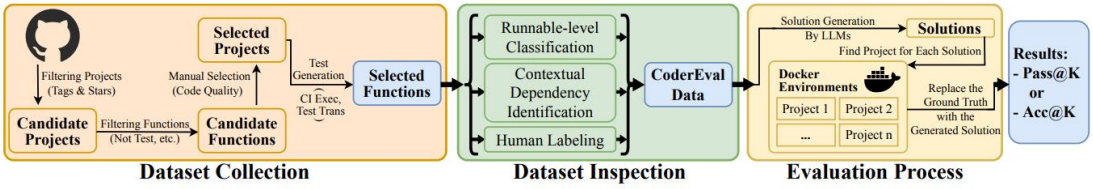


Figure 1: Overview of CoderEval construction process

2.4 评估

作者克隆了所有项目的环境，使用 venv 为每个项目创建一个单独的虚拟环境，使用 pip 安装所有依赖项，并触发项目中的原始测试以验证运行时环境。在环境准备就绪并给定一个要评估的模型后，编写一个程序将生成的代码自动注入到项目中，通过测试输入正确调用函数，并将实际输出与预期输出进行比较。给定模型生成的 n 个代码片段后，程序会用每个代码片段依次替换原始函数。替换并运行后，输出将被捕获。如果函数返回时没有错误或异常，平台会将实际输出与预期输出进行比较，否则，意外终止将被视为测试失败。测试完所有任务的所有代码片段后，所有任务的结果将用于计算最终的生成正确代码的能力（Pass@k）指标。

在 CoderEval for Python 和 CoderEval for Java 上，三种模型生成独立函数的效率都大大高于生成非独立函数的效率。不同的模型在生成代码方面有其独特的能力。

Table 3: Overall effectiveness of three models on two benchmarks

Benchmark	Model	Python			Java		
		Pass@1	Pass@5	Pass@10	Pass@1	Pass@5	Pass@10
CoderEval	CodeGen[1]	9.48%	19.58%	23.48%	13.91%	27.34%	33.48%
	PanGu-Coder[2]	11.83%	20.93%	27.39%	25.43%	37.39%	43.04%
	ChatGPT[3]	21.04%	27.31%	30.00%	35.39%	42.77%	46.09%
HumanEval	CodeGen[1]	10.2%	19.79%	22.8%	5.78%	9.0%	9.45%
	PanGu-Coder[2]	13.42%	21.48%	22.73%	8.21%	15.88%	18.63%
	ChatGPT[3]	39.21%	64.09%	72.96%	38.21%	59.35%	67.23%

[1] We use the 350M CodeGen-Mono model with the default settings for Python. Since CodeGen does not have a monolingual version of Java, on CoderEval for Java, we use the CodeGen-Multi model instead.
[2] PanGu-Coder has two different models for Python and Java, we use the 300M models with the default settings for Python and Java.
[3] We use the "gpt-3.5-turbo" for ChatGPT in our experiments.

大型语言模型生成正确上下文信息的能力（Acc@k）与生成正确代码的能力（Pass@k）高度相关。大语言模型生成不同类型（即 TypeReference、APIInvocation 和 VariableReference）上下文标记的能力因语言而异。

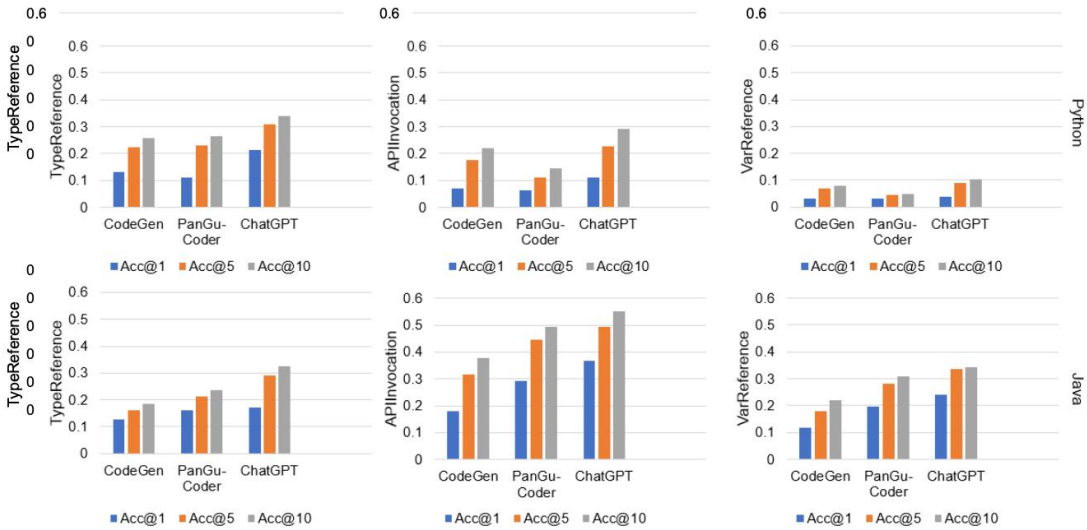


Figure 4: The accuracy of three models that they can correctly generate the contextual tokens

选择使用人工标注的文档字符串还是原始文档字符串会影响代码生成的效果。在单一语言的代码生成任务中，当给出语义相同但表达方式不同的自然语言描述时，使用单一语言语料库训练的模型比使用多种语言训练的模型表现更好。

Table 6: Effectiveness with two Prompt Versions of CoderEval

Model	Prompt	Python			Java		
		Pass@1	Pass@5	Pass@10	Pass@1	Pass@5	Pass@10
CodeGen	Original	9.48%	19.58%	23.48%	13.87%	27.12%	33.04%
	Human Label	12.26%	22.49%	25.65%	10.65%	21.36%	26.52%
PanGu-Coder	Original	11.83%	20.93%	27.39%	25.43%	37.39%	43.04%
	Human Label	13.74%	21.14%	24.78%	26.70%	40.33%	46.09%
ChatGPT	Original	21.13%	27.31%	30.00%	35.39%	42.77%	46.09%
	Human Label	26.61%	31.31%	32.61%	26.96%	34.85%	37.39%

2.5 小结

该文章指出了现有代码函数测试基准的局限性：HumanEval、AiXBench 等现有基准通常只包括独立函数，而在开源项目中，非独立函数占函数的 70%以上。因此，该文章提出了一个实用代码生成基准 CoderEval，其源自不同领域的开源项目，考虑了非原始类型、第三方库和项目特定的上下文引用。此外，CoderEval 还包括生成中函数的人工标注文档字符串，以补充原始文档字符串。

作者在 CoderEval 上评估并比较了三种最先进的代码生成模型（CodeGen、PanGu-Coder 和 ChatGPT）。实验结果发现：这些模型在非独立函数上的效果不如在独立函数上的效果好；对于所有这些模型来说，即使是 ChatGPT（功能最强大的模型），生成具有上下文依赖性的代码也很重要，但却很有挑战性；选择使用人工标注的文档字符串还是原始文档字符串会影响代码生成的效果。

3 代码生成基准数据集研究

论文《AixBench: A Code Generation Benchmark Dataset》[21]发布于 CORR 2022，其提出了一种检测代码生成效果所使用的数据集，包含两个部分，一个是自动化测试数据集，另一个是手动评估的 NL 任务描述数据集。提出了自动化评估代码正确性的新指标，以及手动评估生成代码整体质量的标准。

3.1 研究问题

近年来，出现了利用机器学习技术，特别是基于 Transformer 的大型生成式预训练语言模型进行代码生成辅助的趋势，用户通过给定描述所需功能功能的自然语言描述，利用这些模型生成独立函数（即仅调用或访问内置函数和标准库的函数）以及非独立函数。

为了对现有的生成式预训练模型所生成的代码质量进行量化评估，业界广泛选择了使用代码评估测试数据集进行代码质量评估。本文提出了一个基准数据集，用于评估代码生成模型，这些模型以自然语言为输入，以代码为输出。基准数据集包括 175 个样本用于自动化评估，161 个样本用于手动评估。同时提出了一个新的自动化评估生成代码正确性的指标，以及一套用于手动评估生成代码整体质量的标准。

3.2 相关工作

现有指标局限性：传统指标如精确匹配、BLEU 或困惑度不适用于方法级代码生成的正确性评估。

相关数据集：本文对比了 HumanEval、APPS[23]、PandasEval[24]、NumpyEval[24]、CONCODE[25]和 PY150 等数据集，并指出了它们的局限性：首先，HumanEval 数据集是纯 Python 的；第二，主要是关于纯算法和字符串操作，它们只是现实世界问题的一小部分。第三，HumanEval 中的提示在“文本代码”交互场景中与人类编写的有较大误差等。

基于这些工作，可以发现目前缺乏包含 Java 代码，且自然语言描述部分包含英语和中文的自动化测试数

据集和 NL 任务描述数据集。

3.3 研究方法

本文提出了一个用于自动评估代码生成模型生成代码正确性和手动评估整体质量的基准数据集，用于评估代码生成模型，这些模型以自然语言为输入，以代码为输出。其包含两个部分，一个是自动化测试数据集（175 个样本），另一个是手动评估的 NL 任务描述数据集（161 个样本），其在特定方面优于现有的自动化测试数据集和 NL 任务描述数据集。

自动化测试数据集：从开源项目中挑选的“方法注释-方法实现”对，满足独立性、合理性和描述性的标准。

NL 任务描述数据集：与自动化测试数据集类似，但允许一定程度的模糊性，更接近实际代码中的方法描述。

```
{
    "raw_nl": "Close Reader. If object is null it is ignored",
    "signature": "public static void close(Reader reader)"
}
{
    "raw_nl": "max() that works on three integers",
    "signature": "public static float max(float a, float b, float c) "
}
{
    "raw_nl": "将 Date 类型转为时间字符串，格式为 format",
    "signature": "public static String date2String(final Date date, final DateFormat format)"
}
{
    "raw_nl": "获取类上具有指定注解的接口的名称，如果有多个，则以第一个为准 找不到符合条件的接口则返回 clazz 类的名称",
    "signature": "public static String getInterfaceName(Class<?> clazz, Class<? extends Annotation> annotation)"
}
```

在生成式预训练语言模型执行代码生成任务时，针对其生成的最终代码，本文设置了如下几种评估标准

正确性：根据实现的功能完整性进行评分。

```
{
    "raw_nl": "return the last day of the date's month of specified string value in format: yyyy-MM"
}
{
    "raw_nl": "transform a string to a valid classname string"
}
{
    "raw_nl": "从 http 服务拉取并解析 Properties 文件"
}
{
    "raw_nl": "创建简单颜色选择板"
}
```

代码质量：考虑性能和代码异味。

可维护性：评估代码的标准化程度、可读性和结构清晰度。

三大评分标准均具有由低至高的评分层级，代表质量的提升。针对代码正确性，有(a) 4 分：指定的功能已完全实现；(b) 3 分：主要功能已实现。但是，缺少了一些细节，这些细节的缺失并不影响整体逻辑的正确性。需要进行一些小的修改以满足所有要求；(c) 2 分：仅实现了核心功能。大部分要求并未在代码中体现。需要更多的修改以满足要求；(d) 1 分：指定的功能完全没有实现。针对代码质量，有(a) 3 分：细节处理得当。在性能方面不存在明显更优的代码。如果可能，资源已得到相应的释放。没有明显的代码异味；(b) 2 分：一些细节处理不当。存在低严重性的代码异味；(c) 1 分：在性能方面存在显著更优的解决方案。或者存在严重的代码异味。针对可维护性，有(a) 5 分：该方法的实现非常标准化，变量命名语义上直接明了，方法没有不必要的冗余，可读性好，代码简短，代码块结构清晰；(b) 4 分：该方法的实现相对标准化，变量命名基本上语义上直接明了，并且可读性较好；(c) 3 分：该方法的实现符合一定的规范，但一些变量名没有意义，并使

用了有缺陷的代码和废弃的方法；(d) 2 分：代码编写方式混乱，或者没有遵循一致的规范，或者变量命名中有很多无意义的名称，或者存在某些重复和冗余的代码。可读性差；(e) 1 分：非常混乱，完全不合逻辑，难以阅读的代码。

可以看出，作者针对生成代码的质量做出了复杂且完善的量化评估标准，可以有效帮助我们在针对生成式预训练语言模型所生成的代码进行评估时确定其质量水平，并明确各大模型在生成代码时存在的特点与优劣之处，推动模型融合合作的可能研究。

3.4 评估

除了 pass@1 (即 k=1 时的 pass@k 的特殊情况) 之外，作者还使用了平均测试用例通过率 (AvgPassRatio) 来评估代码生成模型。AvgPassRatio 的计算方式如下：

$$AvgPassRatio = \frac{1}{n}\sum_{i}^{n} PassRatio_i$$

$$PassRatio_i = \frac{Count_{i,pass}}{Count_{i,total}}$$

评估指标：使用 pass@1 和 AvgPassRatio 来评估代码生成模型。

模型比较：对 aiXcoder XL[26]和 GitHub Copilot[27]进行了评估，并比较了它们在自动化测试和手动评估上的表现。

Metrics	aiXcoder XL	Copilot
pass@1	86	81
	(49.14%)	(46.29%)
AvgPassRatio	120.1979	121.7152
	(68.68%)	(69.55%)

Table 2: Automatic Comparison on Correctness over 175 samples

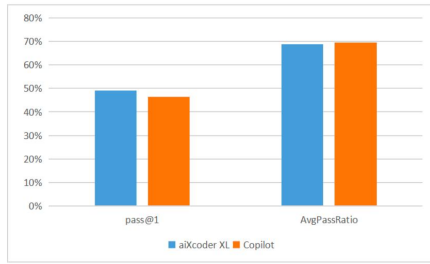


Figure 3: Automatic Comparison on Correctness

Metrics	aiXcoder XL	Copilot
Correctness	2.9503	2.9875
Code Quality	2.2049	2.1739
Mantainability	2.9937	3.0931

Table 3: Manual Comparison between aiXcoder XL and Copilot on NL Task Description Dataset

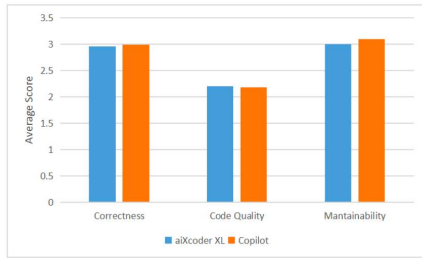


Figure 4: Manual Comparison between aiXcoder XL and Copilot on NL Task Description Dataset

3.5 小结

本文提出了一个基准数据集 AixBench，用于评估代码生成模型，这些模型以自然语言为输入，以代码为输出。该数据集包含两个部分，一个是自动化测试数据集，另一个是手动评估的 NL 任务描述数据集。以此提出了自动化评估代码正确性的新指标，以及手动评估生成代码整体质量的标准。

同时，本文还评估了两个已发布的代码生成产品，aiXcoder XL 和 GitHub Copilot 的表现，其结果相似。

4 总 结

生成式预训练语言模型在代码生成、日志管理环节拥有长足的进步，然而，其关键技术仍处在探索阶段，研究者缺乏对于预训练语言模型工作原理的认识，对于其输出结果的评价标准也参差不齐，无法形成统一的量化标准，并且在生成代码与日志理解层面仍存在过程优化、性能提升的空间。目前，许多项目代码中都存在生成式预训练语言模型参与的痕迹，其代码质量与日志调优能力直接影响了软件项目开发与维护的质量，影响着潜在成本。为了能够更好地针对生成式预训练语言模型在代码质量分析量化与日志理解调优进行量化评估与性能提升，相关领域的研究一直是软件测试领域的一大重要话题。

本文整理的第一篇论文针对生成式预训练语言模型在日志理解领域的实际情况。提出了知识增强的日志理解预训练语言模型 KnowLog，以此提高日志理解任务的最先进性能，作者通过并提出了三个新的预训练任务，有效地利用领域知识来增强模型效果，使日志的表示更加通用。

在第二篇论文中，作者主要对于现有的生成式预训练语言模型在生成非独立函数（即实际应用生产环境的函数）时的能力进行深入研究，提出了 CoderEval，一个上下文感知的基准测试，它可以用于评估在实用代码生成方面的代码生成模型。CoderEval 可以为在实用代码生成任务中代码生成模型的有效性提供更实际和具有代表性的评估。该方法是迄今为止唯一支持项目级代码生成并使用生成正确代码能力（Pass@k）评估指标的基准，可以验证生成代码的功能正确性。

第三篇论文主要作为第二篇论文的前置研究，是一套检测代码生成效果所使用的数据集，包含两个部分，一个是自动化测试数据集，另一个是手动评估的 NL 任务描述数据集。提出了自动化评估代码正确性的新指标，以及手动评估生成代码整体质量的标准。该系列论文在这方面的研究普遍存在测试集规模较小、过于依赖独立函数，缺乏对非独立函数评估数据的缺陷，由此启发了第二篇论文作者的重点研究内容。

References:

[1] Ma L, Yang W, Xu B, et al. KnowLog: Knowledge Enhanced Pre-trained Language Model for Log Understanding[C]//Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. 2024: 1-13.

[2] Huang S, Liu Y, Fung C, et al. Hitanomaly: Hierarchical transformers for anomaly detection in system log[J]. IEEE transactions on network and service management, 2020, 17(4): 2064-2076.

[3] Li X, Chen P, **g L, et al. Swisslog: Robust and unified deep learning based log anomaly detection for diverse faults[C]//2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2020: 92-103.

[4] Chen Z, Gao Q, Moss L S. NeuralLog: Natural language inference with joint neural and logical reasoning[J]. arxiv preprint arxiv:2105.14167, 2021.

[5] Egersdoerfer C, Zhang D, Dai D. Clusterlog: Clustering logs for effective log-based anomaly detection[C]//2022 IEEE/ACM 12th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS). IEEE, 2022: 1-10.

[6] Nedelkoski S, Bogatinovski J, Acker A, et al. Self-supervised log parsing[C]//Machine Learning and Knowledge Discovery in Databases: Applied Data Science Track: European Conference, ECML PKDD 2020, Ghent, Belgium, September 14 – 18, 2020, Proceedings, Part IV. Springer International Publishing, 2021: 122-138.

[7] Tao S, Meng W, Cheng Y, et al. Logstamp: Automatic online log parsing based on sequence labelling[J]. ACM SIGMETRICS Performance Evaluation Review, 2022, 49(4): 93-98.

[8] Le V H, Zhang H. Log parsing with prompt-based few-shot learning[C]//2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023: 2438-2449.

[9] Guo H, Lin X, Yang J, et al. Translog: A unified transformer-based framework for log anomaly detection[J]. arxiv preprint arxiv:2201.00016, 2021.

[10] Qi J, Luan Z, Huang S, et al. Logencoder: Log-based contrastive representation learning for anomaly detection[J]. IEEE Transactions on Network and Service Management, 2023.

[11] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need[J]. Advances in neural information processing systems, 2017, 30.

[12] Devlin J, Chang M W, Lee K, et al. Bert: Pre-training of deep bidirectional transformers for language understanding[J]. arxiv preprint arxiv:1810.04805, 2018.

[13] Yu H, Shen B, Ran D, et al. Codereval: A benchmark of pragmatic code generation with generative pre-trained models[C]//Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. 2024: 1-12.

[14] Chen M, Tworek J, Jun H, et al. Evaluating large language models trained on code[J]. arxiv preprint arxiv:2107.03374, 2021.

[15] Li Y, Choi D, Chung J, et al. Competition-level code generation with alphacode[J]. Science, 2022, 378(6624): 1092-1097.

[16] Fried D, Aghajanyan A, Lin J, et al. Incoder: A generative model for code infilling and synthesis[J]. arxiv preprint arxiv:2204.05999, 2022.

[17] Nijkamp E, Pang B, Hayashi H, et al. Codegen: An open large language model for code with multi-turn program synthesis[J]. arxiv preprint arxiv:2203.13474, 2022.

[18] Christopoulou F, Lampouras G, Gritta M, et al. Pangu-coder: Program synthesis with function-level language modeling[J]. arxiv preprint arxiv:2207.11280, 2022.

[19] John Schulman, Barret Zoph, Christina Kim, Jacob Hilton, Jacob Menick, Jiayi Weng, Juan Felipe Ceron Uribe, Liam Fedus, Luke Metz, Michael Pokorny, et al. 2022. ChatGPT: Optimizing Language Models for Dialogue. OpenAI blog (2022).

[20] Lai Y, Li C, Wang Y, et al. DS-1000: A natural and reliable benchmark for data science code generation[C]//International Conference on Machine Learning. PMLR, 2023: 18319-18345.

[21] Hao Y, Li G, Liu Y, et al. Aixbench: A code generation benchmark dataset[J]. arxiv preprint arxiv:2206.13179, 2022.

[22] Cassano F, Gouwar J, Nguyen D, et al. MultiPL-E: a scalable and polyglot approach to benchmarking neural code generation[J]. IEEE Transactions on Software Engineering, 2023.

[23] Hendrycks D, Basart S, Kadavath S, et al. Measuring coding challenge competence with apps[J]. arxiv preprint arxiv:2105.09938, 2021.

[24] Zan D, Chen B, Yang D, et al. CERT: continual pre-training on sketches for library-oriented code generation[J]. arxiv preprint arxiv:2206.06888, 2022.

[25] Iyer S, Konstas I, Cheung A, et al. Map** language to code in programmatic context[J]. ar** language to code in programmatic context." ar** language to code in programmatic context. arxiv preprint arxiv:1808.09588.

[26] aiXcoder. aixcoder xl natural language to code demo.

[27] GitHub. Github copilot, your ai pair programmer.