软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.000000]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

重新审视代码补全中的检索增强策略*

邹佰翰[1,2], 汪莹[1,2], 彭鑫[1,2], 娄一翎[1,2], 刘力华[3], 张昕东[3], 林帆[3], 刘名威[1,2]

[1](复旦大学 计算机科学技术学院, 上海 200438)

[2](上海市数据科学重点实验室(复旦大学), 上海 200438)

[3](阿里巴巴集团, 浙江 杭州 310030)

通讯作者: 刘名威, E-mail: liumingwei@fudan.edu.cn

摘 要: 软件开发者在编写代码时,常常会参考项目中实现了相似功能的代码. 代码生成模型在生成代码时也具有类似特点，会以输入中给出的代码上下文信息作为参考. 基于检索增强的代码补全技术与这一思想类似，该技术从检索库中检索到外部代码作为额外信息，对生成模型起到提示的作用，从而生成目标代码. 现有的基于检索增强的代码补全方法将输入代码和检索结果直接拼接到一起作为生成模型的输入，这种方法带来了一个风险，即检索到的代码片段可能并不能对模型起到提示作用,反而有可能会误导模型,导致生成的代码结果不准确. 此外,由于无论检索到的外部代码是否与输入代码完全相关，都会被与输入代码拼接起来输入到模型，这导致该方法的效果在很大程度上依赖于代码检索阶段的准确性. 如果检索阶段不能返回可用的代码片段，那么后续的代码补全效果可能也会受到影响. 首先，本文针对现有的代码补全方法中的检索增强策略进行了经验研究，通过定性和定量实验分析检索增强的各个阶段对于代码补全效果的影响，在经验研究中重点识别了代码粒度、代码检索方法、代码后处理方法这三种影响检索增强效果的因素. 接着，本文基于经验研究的结论设计改进方法，提出一种通过分阶段优化代码检索策略来改进检索增强的代码补全方法 MAGIC（Multi-stAGe optImization for retrieval augmentated Code completion），设计了代码切分、二次检索精排、模板提示生成等改进策略，可以有效地提升检索增强对代码补全模型的辅助生成作用，并减少模型在代码生成阶段受到的噪声干扰，提升生成代码的质量. 最后，本文在 Java 代码数据集上的实验结果表明: 与现有的基于检索增强的代码补全方法相比，该方法在编辑相似度和完全匹配指标上分别提升了 6.76 个百分点和 7.81 个百分点. 与 6B 参数量的代码大模型相比，该方法能够在节省 94.5%的显存和 73.8%的推理时间的前提下，在编辑相似度和完全匹配指标上分别提升 5.62 个百分点和 4.66 个百分点.

关键词: 检索增强; 大语言模型; 代码补全; 提示学习; 多阶段优化

中图法分类号: TP311

中文引用格式: 邹佰翰, 汪莹, 彭鑫, 娄一翎, 刘力华, 张昕东, 林帆, 刘名威. 重新审视代码补全中的检索增强策略. 软件学报,2023,32(7). http://www.jos.org.cn/1000-9825/9999.htm

英文引用格式: Zou BH, Wang Y, Peng X, Lou YL, Liu LH, Zhang XD, Lin F, Liu MW. Revisiting The Retrieval-Augmentation Strategy In Code Completion. Ruan Jian Xue Bao/Journal of Software, 2023 (in Chinese). http://www.jos.org.cn/1000-9825/9999.htm

Revisiting The Retrieval-Augmentation Strategy In Code Completion

ZOU Bai-Han[1,2], WANG Ying[1,2], PENG Xin[1,2], LOU Yi-Ling[1,2], LIU Li-Hua[3], ZHANG Xin-Dong[3], LIN Fan[3], LIU Ming-Wei[1,2]

[1](School of Computer Science, Fudan University, Shanghai 200438, China)

[2](Shanghai Key Laboratory of Data Science (Fudan University), Shanghai 200438, China)

* 基金项目: 国家自然科学基金(62302099);
收稿时间: 2023-12-22; 修改时间: 2024-3-20; 采用时间: 2024-5-15

[3](Alibaba Group, Hangzhou 310030, China)

Abstract: When writing code, software developers often refer to code snippets that implement similar functions in the project. The code generation model has similar characteristics when generating code fragments, and will use the code context provided in the input as a reference. The external code retrieved from the retrieval library is used as additional context information to prompt the generation model so as to complete the unfinished code fragments. This method is called code completion technology based on retrieval augmentation. The existing code completion method based on retrieval augmentation directly splices the input code and retrieval results together as the input of the generated model. Although this method can provide more context information for the model, it also brings a risk that the retrieved code fragments may not prompt the model, but may mislead the model, resulting in inaccurate or irrelevant code results. In addition, whether the retrieved external code is completely related to the input code or not, it will be spliced with the input code and input to the model, which leads to the effect of this method largely depends on the accuracy of the code retrieval stage. If the retrieval phase cannot return the available code fragments, the subsequent code completion effect may also be affected. This paper conducts an empirical study on the retrieval augmentation strategies in the existing code completion methods. Through qualitative and quantitative experiments, it analyzes the impact of each stage of retrieval augmentation on the effect of code completion. In the empirical study, it focuses on identifying three factors that affect the effect of retrieval augmentation, namely, code granularity, code retrieval methods, and post-processing methods. Based on the conclusion of empirical research, an improved method is designed, and a code completion method MAGIC (multi-stage optimization for retrieval augmented code completion) is proposed to improve the retrieval augmentation by optimizing the code retrieval strategy in stages. The improved strategies such as code segmentation, retrieval-reranking, template prompt generation are designed, which can effectively enhance the auxiliary generation effect of the code retrieval module on the code completion model, and reduce the interference of irrelevant code in the code generation phase of the model, and improve the quality of generated code. The experimental results on Java code dataset show that: compared with the existing code completion methods based on retrieval augmentation, this method improves the editing similarity and perfect matching index by 6.76% and 7.81% respectively. Compared with the large code model with 6B parameters, this method can save 94.5% of the video memory and 73.8% of the inference time, and improve the editing similarity and complete matching index by 5.62% and 4.66% respectively.

Key words: retrieval augmentation; large language model; code completion; prompt learning; multi-stage optimization

在日益复杂和庞大的软件系统中，编写高质量且高效的代码是软件开发过程中的一项重要任务. 然而，由于软件系统的复杂性和代码量的增加，程序员常常面临着编写繁琐和冗长代码的挑战. 为了提高开发效率和质量，编程环境中的代码补全功能应运而生.

代码补全是对程序员正在编写的代码进行辅助，提供与当前上下文相关的代码片段、函数、方法、变量等的建议和自动补全. 它基于已有的代码、语法规则和上下文信息，预测和推断程序员可能要输入的代码，并提供相关的选项和建议[1]. 代码补全功能能够在编码过程中提供即时的帮助，减少代码输入的工作量，提高编码的效率和准确性. 目前许多公司已经意识到代码补全的重要性，并将其集成到其开发工具中. 例如，微软公司的 Visual Studio 和 JetBrains 公司的 IntelliJ IDEA 等流行的 IDE 都提供了强大的代码补全功能. 这些工具通过基于上下文的预测和分析，为程序员提供了准确的代码建议和补全选项. 同时，一些在线代码编辑器和代码托管平台如 GitHub、CodePen 等也在其编辑器中集成了代码补全功能，为程序员提供更好的编码体验.

现有的代码补全方法可以分为基于规则的代码补全方法、基于概率统计模型的代码补全方法、基于深度学习的方法、混合方法. 其中，基于规则的方法依赖于预定义的编程语言语法规则和模式[2]. 它通常使用字符串匹配或者编程语言的抽象语法树（Abstract Syntax Tree, AST）来预测代码的下一部分. 基于规则的方法较为简单，易于实现，但它们的灵活性和泛化能力有限，难以处理复杂的代码补全任务. 基于统计概率模型的方法通过分析大量的代码库，学习代码片段的统计分布. 例如 Raychev 等人[3]提出通过 n-gram 模型记录历史代码数据来预测代码序列的下一个元素. 这种方法通常比基于规则的方法更加灵活，但无法充分挖掘代码中蕴含的特征信息. 基于深度学习的方法通过搭建深度神经网络模型对代码进行建模[4]，利用大规模代码数据集训练模型，使模型学习到程序语言的高维表征. 例如，Bhoopchand 等人[5]利用指针神经网络预测出 Python 代码的

下文内容. 此外，基于 Transformer 的模型如 GPT 和 BERT 在代码补全任务中也取得了显著的进展[6]. 混合方法则结合了以上几种技术的优点，通过集成不同的技术来提高代码补全的性能. 例如基于检索增强代码补全技术，将代码检索与基于深度学习的代码生成方法相结合，有效提升生成代码的可靠性.

检索增强技术由 Lewis 等人[7]提出，通常指利用检索机制来辅助语言模型，增强其处理和生成文本的能力. 这种方法结合了传统的检索系统（如搜索引擎）和最新的深度学习技术. 检索增强技术的核心思想是利用外部的知识源（例如大型数据库或互联网）来提供上下文信息，帮助模型在处理复杂任务时做出更加准确和丰富的预测. 由于模型在训练阶段可能会存在知识遗忘问题，检索增强的方式能够将一部分知识信息存储在独立的数据库中形成非参数记忆，且可以动态更新检索库的内容而不需要重新训练模型，能够在降低模型参数量的同时，有效控制模型的生成内容，提升推理的速度和准确度.

在软件开发场景下，程序员常常会参考已有代码中的功能实现方法，复制外部的代码粘贴到当前的代码中并进行编辑修改[8]，这一过程与检索增强技术通过检索来引导模型生成的过程有相似之处. 目前已经有研究人员开始探索结合通过检索增强技术来提高代码补全的质量和效率. 现有的基于检索增强的代码补全技术采用的基本流程是：首先根据输入的未完成代码片段，从预先构建的代码检索库中检索出相似的代码片段；然后对检索出的代码片段，按照其相似度分数进行排序，选出相似度最高的候选结果；接着将检索到的相似代码片段与输入的未完成代码片段拼接在一起；最后将拼接后的代码输入给预训练的生成模型，得到代码补全输出结果. 例如，微软提出了 ReACC 检索增强代码补全框架[9]，该方法通过将向量检索与字符串检索混合加权的方式检索得到与输入代码相关的代码片段，然后将检索结果与输入代码拼接起来输入给生成模型. 该研究通过实验评估发现基于检索增强来引导模型生成的代码补全方式在效果上要好于直接使用生成模型.

虽然现有的基于检索增强的代码补全技术取得了不错的效果，但是仍存在一些尚未探索的问题. 一方面，现有的代码补全技术中的检索增强策略，都是在检索到代码结果之后，直接将检索结果与输入代码拼接起来输入给模型，而没有做更多的规则分析和后处理；另一方面，例如对于检索库所保存的代码粒度对模型生成效果的具体影响，目前没有相关研究进行明确的实验对比. 例如，用 GitHub Java Corpus 数据集同时构建出检索库 A 和检索库 B，其中检索库 A 保存完整的 Java 方法代码，检索库 B 保存的是将所有代码按照空格分割为长度不超过 10 个 token 的片段级代码. 使用微软提出的 ReACC 检索增强补全方法在检索库 A 和检索库 B 上同时测试，尝试补全红色下划线部分的代码下文，得到的测试结果如图 1 所示，可以看出从同一个数据集构建的不同粒度的代码库，得出了不同的代码补全结果. 这说明代码检索库保存的代码粒度的配置对于代码补全的结果可能存在一定的影响. 而现有研究中仅仅是直接使用未经过处理的数据集进行代码检索，没有考虑具体的代码粒度大小的设计.

在目前普遍将检索增强与大语言模型结合的场景下，已经有一部分自然语言处理领域的研究工作探索了不同的因素对于检索增强效果的影响. 例如，Borgeaud 等人[10]通过扩充检索库规模，构建了一万亿级别的超大规模检索库来提升文本生成的效果；Chen 等人[11]提出先将维基百科文本分割为子命题，再将子命题作为检索结果提示模型完成知识问答，该方法的检索增强效果优于直接使用完整文本或段落进行问答；Ram 等人[12]尝试在不微调模型的情况下将检索结果与提示模板结合，引导问答模型生成高质量回答，发现效果要优于直接将检索结果与问题拼接的方法.

但是在代码补全领域，目前类似的分析代码补全中的检索增强策略的影响因素的研究工作比较少. 因此，我们决定重新审视检索增强的代码补全技术中的代码检索策略. 具体来说，我们重点识别了检索增强过程中的三个因素进行探索：检索库的代码粒度（因素 a）、代码检索的方法（因素 b）以及代码检索结果的后处理方法（因素 c）.

我们针对以上三个因素分别进行了经验研究实验的探索. 对于因素 a，我们从检索增强效果、资源开销等指标上对比了不同配置的代码检索库对于检索增强代码补全技术的影响. 我们通过实际实验发现，当代码检索库的规模达到 300 万及以上时，向量检索的内存开销和检索所需的时间成本会大大上升，从而影响代码补全的实际速度. 这也启发我们设计新的检索方式在兼顾检索增强效果的同时降低时空开销. 对于因素 b，我们

尝试将几种不同功能的代码作为检索结果引导模型生成，发现检索增强的方式并不总是能够引导模型生成正确的内容，低质量代码可能不但无法对生成模型起到提示作用，反而成为模型的噪声. 对于因素 c，主要探索如何结合检索结果与输入代码，我们对比了直接将检索结果与输入代码拼接在一起，和使用特殊 token 先分隔两段代码再进行拼接的方式，发现直接进行代码拼接的方式的检索增强效果在指标上要差于使用特殊 token 分隔的方法，说明缺少明确分隔界限的代码拼接有可能会导致生成模型对上下文信息的混淆. 对上述三个因素的经验研究实验启发我们从代码检索库构建、检索排序、检索结果后处理阶段分别改进现有方法.

	检索库A（方法级代码）	检索库B（片段级代码）

原输入代码	public Component inflate(InputStream xml) throws LayoutInflaterException{
	LayoutInflaterContentHandler handler = new LayoutInflaterContentHandler(componentManager);
	XMLReader parser = XMLReaderFactory.createXMLReader();
	parser.setContentHandler(handler);
	parser.parse(new InputSource(xml));
	return handler._____	

检索结果	public Component inflate(InputStream xml, List<ComponentProvider> componentProviders) {
	LayoutInflaterContentHandler contentHandler = new LayoutInflaterContentHandler(componentProviders);
	XMLReader parser = XMLReaderFactory.createXMLReader();
	parser.setContentHandler(contentHandler);
	parser.parse(new InputSource(xml));
	return contentHandler.getLayoutRoot();
	}
		parser.setContentHandler(contentHandler);
		parser.parse(new InputSource(xml));
		return contentHandler.finalizeInflation().setInitialState().createComponentHierarchy();
		}

补全结果	getLayoutRoot();	finalizeInflation().setInitialState().createComponentHierarchy();
真实值	getLayoutRoot();	

图 1 基于不同代码粒度的检索库进行检索增强后的输出结果案例

基于重新经验研究的结果，本文提出了通过分阶段优化代码检索策略来改进检索增强的代码补全的方法，该方法主要分为三个阶段：检索库预构建阶段、检索精排阶段、提示生成阶段. 在检索库预构建阶段，按照代码长度将 1480 万条 Java 方法切分为 3872 万个子代码片段，提升检索库代码的多样性，并利用索引存储的方式保存检索库代码片段的上下文信息. 在检索精排阶段，通过 BM25 算法初步检索匹配代码，基于字符串结构相似度和语义相似度的倒秩融合分数进行二次精排序，提取出评分最高的检索结果候选项，并使用语义相似度评估候选结果的可用性，确定具体的代码补全策略；在提示生成阶段，将检索结果与输入的未完成代码片段合并到提示模板中，基于经过提示模板训练的代码生成模型，生成代码补全的结果. 最后，我们通过消融实验证明了每一个模块的优化方法的有效性.

本文方法利用参数量为 124M 的 CodeGPT 模型在 GitHub Java Corpus 数据集上取得了编辑相似度（Edit Similarity，ES）指标为 70.26、完全匹配（Exact Match，EM）指标为 42.65 的效果，综合性能接近参数量为 16B 的 CodeGen 大模型，且单条代码补全推理速度为该 16B 模型的 4.94 倍，所需显存开销仅为 16B 大模型所需显存开销的 3.5%. 这说明我们的方法能够在提升代码补全性能的同时有效限制代码补全方法带来的时空开销.

本文工作的主要贡献如下：

- 通过经验研究分析了代码补全中的检索增强策略的可改进探索的方向. 本文重新审视了现有检索增强生成技术存在的问题，重点识别了代码粒度、代码检索方法、代码后处理方法三个因素对检索增强和代码补全效果的影响，并设计半定性半定量实验，分析现有的基于检索增强的代码生成技术存在的问题与不足，从多个角度提供了可供未来进一步探索的检索增强的改进思路.
- 基于经验研究的结论提出了一种代码补全技术中的检索增强策略的改进方法 MAGIC. 通过代码切分

来扩充数据库规模，并以索引的方式保留代码片段之间的上下文顺序关系，结合 BM25 粗排+倒秩重排+提示生成的方式，为代码补全模型的提供符合当前代码上下文的提示信息.

- 本文在阿里巴巴企业级生产环境下进行代码补全效果对比、改进方法的消融实验和与代码大模型的性能开销对比这三组实验，证明了基于经验研究提出的改进方法的有效性. 实验结果表明我们优化后的模型能够在 Java 数据集上将现有的基于检索增强的代码补全模型的编辑相似度和完全匹配指标分别提高 6.76 个百分点和 7.81 个百分点，并在节约 94.5%的显存和 73.8%的推理时间的情况下，在 124M 的 CodeGPT 模型上相比于 6B 的 CodeGen 代码大模型分别提 5.62 个百分点和 4.66 个百分点.

本工作的创新点在于，针对代码补全场景下的检索增强的影响因素进行了更完善的探索和针对性优化，并提供了未来改进代码补全中的检索增强策略的思路，弥补了目前代码领域针对检索增强相关影响因素研究的空白.

本文第 1 节介绍代码领域与检索增强相关的工作. 第 2 节对现有的检索增强技术存在的问题及其各个可优化阶段进行了重新审视与半定性半定量分析. 第 3 节介绍了本文所述的检索增强各个阶段的具体优化方法. 第 4 节介绍了本文设计的 4 个评估实验及其实验结果. 第 5 节对基于检索增强的代码补全技术进行了总结与展望.

1 代码领域的检索增强相关工作

在软件工程领域的现有研究中，检索增强技术已经被应用在代码补全、代码生成、代码摘要生成、代码提交信息生成的场景上. 这些技术通过检索和使用大规模检索库中的数据来辅助代码或文本信息的自动补全和生成过程，其生成效果要好于直接使用微调后的模型.

1.1 基于检索增强的代码补全

代码补全任务的目的是对程序员正在编写的代码进行辅助，提供与当前上下文相关的代码片段、函数、方法、变量等的建议. 代码补全任务的输入是一个未完成的代码片段，输出是该代码片段对应的下文.

Luan 等人[13]利用结构化代码搜索在大型代码库中搜索与输入代码片段结构上相似的方法体，为用户提供多个推荐选项. 该方法先对包含数千个开源项目的大型代码库建立索引，给定一个代码片段作为输入查询后，在代码检索库中搜索包含该代码片段的方法体，并对搜索结果进行聚类和交叉，返回一组推荐的方法体. Lu 等人[9]提出了基于检索增强的代码补全框架 ReACC，该框架结合了源代码检索器和 CodeGPT 自回归语言模型，通过从代码数据库检索相似代码，使用余弦相似度与 BM25 算法的加权结果为检索到的代码片段打分，并将选出的最终检索结果与未补全代码拼接作为代码生成模型的输入. Zhang 等人[14]提出了一种仓库级别的代码补全方法，先仅使用未补全的代码执行代码检索，并生成一种代码的中间表示形式；接着，利用中间代码执行第二次检索迭代，生成最终补全代码. 这一设计证明了利用中间表示形式进行检索增强的可行性. 我们在改进现有方法时参考了这一思想，通过代码切分的方式构造检索库，第一次检索出代码片段后，再通过代码之间的索引进一步获得更多的下文代码片段进行拼接，有效增加输送给模型的代码下文信息.

1.2 基于检索增强的代码生成

代码生成任务是指根据特定的需求或规则，自动生成代码片段或完整的代码. 代码生成任务的输入是对于需求或功能的自然语言描述，输出是需求或描述对应的代码，可以包括类、方法、参数等.

Hashimoto 等人[15]提出了先检索后编辑的思想，即构建了一个检索器和一个编辑器，首先将检索器根据输入的自然语言描述，从训练集中检索相似训练样本，然后编辑器根据检索到的代码，将其编辑为所需的输出代码. Hayati 等人[16]提出首先基于编辑距离的相似度计算与输入描述的结构相似度最高的自然语言描述片段，然后基于抽象语法树和 n-gram 概率模型产生对应代码. 该研究采用的编辑距离相似度能够有效评估两段字符串的结构相似度. Xu 等人[17]提出在引入外部检索库的同时，利用检索库数据对生成模型进行微调，该方法有效提高了模型的代码生成能力. 通过将检索模型和推理能力更强的生成模型结合能够达到更好的代码生成效

果. Parvez 等人[18]提出对于给定的代码功能描述先检索前k个候选代码，然后将候选代码聚合起来，再基于生成器模块（PLBART 模型）生成目标序列. 但这种方式所能拼接的候选代码的长度和数量k受限于模型所能接受的最大输入长度，且没有考虑候选代码可能会作为噪声影响模型生成. 我们的方法通过对前k个候选代码进行重新排序来筛选出最终的检索结果，并利用相似度阈值过滤掉无关代码片段，一定程度上避免了候选代码可能会作为噪声影响模型生成的问题.

1.3 基于检索增强的代码摘要生成

代码摘要生成任务是指根据给定的代码，自动生成其摘要或概要信息的任务. 代码摘要是对代码进行简要描述或总结，以便开发人员快速了解代码的功能、结构或执行流程. 代码摘要生成任务的输入是代码片段，输出是对代码的自然语言描述.

Liu 等人[19]提出用抽象语法树将代码转为代码属性图的形式，利用图结构的代码进行代码检索，并添加检索到的代码以及相应的摘要作为训练模型的辅助信息，提高模型生成摘要的能力. 但图结构的计算复杂度更高，需要耗费大量的时间和计算资源[20]. Li 等人[21]提出先从代码库中检索相似代码片段，然后将原型摘要中的模式与输入代码片段的语义信息结合起来，复用原型摘要中的模式化单词，并根据输入代码的语义信息生成最终的摘要. Yu 等人[22]根据语义和词汇相似度从代码语料库中检索与目标代码最相似的代码，然后使用经过训练的编码器生成两个向量表示并进行融合，最后通过解码器生成代码摘要. 我们的方法也参考了本研究中使用向量计算的思路，利用深度学习模型将代码转化为向量的形式，并计算向量之间的余弦相似度来判断两段代码的相似程度.

1.4 基于检索增强的代码提交信息生成

代码提交信息生成任务是指根据代码变更内容自动化生成版本控制系统提交信息（Commit Message）的任务. 在软件开发中，每次对代码进行更改并提交到版本控制系统时，开发人员通常需要编写一条有意义的提交信息来解释该更改的目的、内容和影响.

Liu 等人[23]使用将代码 diff 转换为抽象语法树的路径进行编码，再输入生成模型产生代码提交信息；并利用代码 diff 匹配策略在训练数据集中检索最相关的代码提交信息，并利用训练好的深度学习模型对两种方式产生的候选项进行打分排序. Wang 等人[24]提出在模型生成阶段使用预先检索出的相似代码 diff 作为辅助信息，提示模型生成高质量的代码提交信息. 这种方式与基于检索增强的代码生成框架 ReACC 的思想相似，但模型没有经过训练来学习如何有效地利用检索结果的信息. Shi 等人[25]维护了一个包含代码 diff 与代码提交信息的检索库，并预先训练了一个用于理解代码 diff 语义的编码器. 首先将检索器检索出的代码 diff 和代码提交信息与原始的代码 diff 一同输入编码器转为向量形式并进行融合，然后利用解码器生成出代码 diff 对应的代码提交信息. 这种基于检索增强生成代码提交信息的方式将检索过程也加入到模型的训练过程中，生成效果要好于直接将代码 diff 输入到生成模型的效果. 我们设计的改进方法也在模型的训练阶段针对每段输入代码进行检索，然后将检索结果与输入代码组合起来作为训练数据. 这种方式训练出的模型的代码补全效果要好于仅仅在模型推理阶段添加检索增强信息的方法.

2 基于检索增强的代码补全经验研究

为了确定基于检索增强的代码生成技术可能存在的优化方向，我们将基于检索增强的代码补全技术划分为检索前、检索中、检索后生成前，这三个阶段. 其中，“检索前”代表的是在进行代码检索之前的代码库构建、代码预处理，在后文中将这一阶段的内容统称为检索预处理阶段；“检索中”代表的是在检索代码时采用的代码检索策略，在后文中称为代码检索阶段；“检索后生成前”代表的是将检索到的代码拼接到输入内容上时，对输入代码和模型进行的前处理，在后文中称为检索后生成前阶段. 我们重点识别了这三个阶段对应的三个因素进行经验研究实验：

- 因素 a：检索库的代码粒度对基于检索增强的代码补全技术的影响. 对于因素 a，为了探究检索库中不

同的代码粒度对代码补全结果的影响，我们在不改变检索增强技术的基本方法的情况下，尝试对比了几种不同代码粒度、不同数据规模的代码检索库，观察其对代码补全效果的影响.

- 因素 b：代码检索的方法对基于检索增强的代码补全技术的影响. 对于因素 b，为了探究不同的检索方法对代码补全效果的影响，我们尝试使用几种不同的代码检索方法，将检索到的几段不相同的代码作为检索结果输入给模型，并对比不同的检索结果对代码补全效果的影响.
- 因素 c：代码检索结果的后处理方法对基于检索增强的代码补全技术的影响. 对于因素 c，为了探究不同的后处理方法对代码补全效果的影响，我们对比了直接将检索结果与输入代码拼接在一起，和使用特殊 token 先分隔两段代码再进行拼接的方式下的代码补全结果.

2.1 检索库的代码粒度对基于检索增强的代码补全技术的影响（因素a）

现有的基于检索增强的代码补全技术所使用的检索库，依照检索库所存储代码的数据类型划分，有字符串文本和数值向量两种类型. 在此基础上，这两种类型各自还包括方法级和文件级两种粒度. 对于文本类型的代码库来说，方法级代码检索库存储的是被解析为完整的方法体或函数的形式的代码；文件级代码检索库是直接基于源代码文件构建而来，存储的是字符串类型的代码片段，既可以是完整的代码体或类体，也可以是代码文件的切片；对于数值向量类型的代码库来说，方法级代码检索库和文件级代码检索库是利用深度学习嵌入模型将对应的文本类型代码库转化为数值向量形式得来的，这些向量通常会捕捉代码的语法结构、语义信息和上下文关系[26].

为了探究文本和向量这两种类型的检索库各自对应的两种不同的代码表征粒度对基于检索增强的代码补全技术的影响，我们从开源代码库和代码数据集中抽取代码，分别构建出文本和向量类型各自对应的方法级代码检索库和文件级代码检索库，并根据基于检索增强的代码补全技术的基本框架进行对比实验. 文本和向量类型的代码检索库的基本配置如下：

- **文本类型代码检索库**：检索库基于 ElasticSearch 实现. 我们将从开源平台收集到的源代码直接存储在 ElasticSearch 中，构建文件级代码检索库；我们使用 JavaParser 解析工具从代码中抽取出方法体存储在 ElasticSearch 中，构建方法级代码检索库，通过 BM25 算法搜索相似代码.
- **向量类型代码检索库**: 在本文经验研究的实验中，向量类型检索库基于 ElasticSearch 和阿里 Proxima 向量检索工具[27]实现. 我们利用 Sentence-BERT 模型将文本类型的方法级和文件级代码库中的代码映射到 768 维的 embedding 向量，将向量存储在 ElasticSearch 中. 在检索向量类型代码时，ElasticSearch 将计算向量间的距离，检索得到与查询代码的向量距离最近的结果. 其中，Sentence-BERT 模型是 Reimers 等人[28]提出的一种向量表征模型，目前已经被广泛应用在向量类型代码检索方面.

在代码数据来源方面，我们准备了三种不同的数据规模的代码数据集，并基于这三种不同数据规模的代码库，都分别构建出文本和向量两种类型所对应的方法级和文件级代码检索库，即共构建出十二个不同配置的检索库. 其中三种不同的数据规模配置如下：

- **小规模代码库**：由从 CodeXGLUE 数据集的训练集中收集的 9.4 万条 Java 代码构成. 我们将这 9.4 万条 Java 代码处理为不同的粒度，分别存储在文本和向量两种类型所对应的方法级和文件级代码检索库中.
- **中等规模代码库**：由从 StackOverFlow 帖子中收集的 34.6 万条 Java 代码构成. 存储形式同上.
- **大规模代码库**：由我们从 GitHub 平台共 45,196 个开发者的 145,532 个代码仓库中收集的 968 万条 Java 代码构成. 存储形式同上.

我们根据基于检索增强的代码补全技术的基本框架，在 CodeXGLUE 数据集的测试集上进行对比实验. 其中，检索功能基于 ElasticSearch 实现，文本类型的代码检索使用 ElasticSearch 默认的 BM25 算法，向量类型的代码检索使用 Proxima 引擎默认的 HNSW 算法，以余弦相似度衡量向量间的距离. 生成功能基于 CodeGPT 实现. 最终得到的代码补全评估效果如表 1 所示，其中 ES 代表编辑相似度（Edit Similarity），EM 代表完全匹配（Exact Match），Mem 代表构建检索库占用的物理内存（单位为 GiB），T 代表处理单条代码消耗的时间（单

位为秒）.

表 1 不同代码粒度和数据规模下经过检索增强的代码补全结果

类型	文本类型检索库								向量类型检索库							
粒度	方法级				文件级				方法级				文件级			
指标	ES	EM	Mem	T	ES	EM	Mem	T	ES	EM	Mem	T	ES	EM	Mem	T
94K	63.29	36.62	**0.03**	**0.05**	63.22	36.67	0.03	0.05	64.49	36.43	13.42	0.07	63.40	35.91	9.59	0.07
346K	64.86	37.32	0.84	0.19	64.45	36.79	0.84	0.39	64.77	36.94	49.66	0.58	64.28	36.46	31.83	0.45
9680K	**65.82**	**39.56**	14.83	1.17	65.53	37.94	14.83	1.02	65.52	38.09	587.28	2.24	65.07	37.38	391.52	1.96

从表中可以看出，在基于同样的代码来源构建出文本和向量两种数据类型对应的代码检索库后，针对同样的检索增强方法，方法级代码检索库的检索增强代码补全效果整体要好于文件级代码检索库. 在数据规模为 968 万的代码库上，使用文本类型的方法级检索库比使用向量类型的方法级检索库在编辑相似度指标上高出 0.45 个百分点，完全匹配指标则高出 3.8 个百分点，且内存占用量仅为向量类型的方法级检索库内存占用量的 2.53%. 同时，使用代码规模为 968 万条代码的检索库进行检索增强产生的代码补全效果要好于使用代码规模为 9.4 万条代码和 34.6 万条代码的检索库.

此外，在实际实验中，我们发现文本类型代码检索库在实际内存开销上要远小于向量类型的代码检索库. 本文使用具有 1.5T 物理内存和 12 核 CPU 的阿里云企业级服务器维护向量类型的检索库，968 万条代码构建的向量检索库需要占用 587.28G 物理内存，且单条代码检索的时间是文本类型代码检索库所需时间的 1.91 倍，说明在检索过程中向量检索库需要占用大量资源，存在一定的局限性.

启示 1：据此，我们推论出扩大检索库的规模和选择合适的代码表征粒度，可能可以在一定程度上提升检索增强的效果.

2.2 代码检索方法对基于检索增强的代码补全技术的影响（因素b）

在检索阶段，选取不同的检索算法有可能会产生不同的检索结果[29]. 对于文本类型的代码，BM25 算法和 TF-IDF 算法是 ElasticSearch 检索工具常用的两种文本类型检索方法；对于向量类型的代码，HNSW 算法是 ElasticSearch 检索工具常用的向量检索方法，该方法具体可以通过余弦相似度和欧氏距离两种算法计算向量间的相似性. 由因素 a 对应的实验可知方法级代码检索库的检索增强代码补全效果整体要好于文件级代码检索库，因而后续经验研究实验均在方法级检索库上展开. 我们尝试在由 9.4 万、34.6 万、968 万条代码构成的三种数据规模的方法级代码检索库上，对 CodeXGLUE 数据集的测试集代码分别使用 BM25 算法、TF-IDF 算法、余弦相似度算法、欧氏距离算法进行代码检索，然后评估使用该检索方法后的代码补全效果，得到的实验结果如表 2 所示.

表 2 方法级检索库的不同检索算法下检索增强后的代码补全效果

评估指标	文本类型检索库						向量类型检索库					
	TF-IDF			BM25			余弦相似度			欧氏距离		
	94K	346K	9680K	94K	346K	9680K	94K	346K	9680K	94K	346K	9680K
ES	63.55	63.92	64.07	63.29	64.86	**65.82**	64.49	64.77	65.52	63.45	64.36	65.13
EM	35.17	35.95	36.21	36.62	37.32	**39.56**	36.43	36.94	38.09	35.99	36.51	37.49

从表中可以看出，在中等规模代码库和大规模代码库上，基于 BM25 算法实现检索增强的方法在四种检索方法中取得的检索增强效果最好. 其中，在由 968 万条代码构建出的代码检索库上，基于 BM25 算法实现检索增强产生的代码补全效果在两项指标上要比使用 TF-IDF 算法实现的效果分别高出 2.7 个百分点和 9.3 个百分点，比使用余弦相似度的向量检索算法的效果分别高出 0.5 个百分点和 3.9 个百分点，比使用欧氏距离的向量检索算法实现的效果分别高出 0.7 个百分点和 5.5 个百分点.

	BM25	TF-IDF	余弦相似度	欧氏距离

原输入代码	public ServerStatus startServer(int port) {
	ServerSocket server = null;
	server = new ServerSocket(port);
	while (!server.isClosed()) {
	Socket client = server.accept();
	handleClient(client);
	}
	}
	return server._____			

检索结果	public ServerStatus initiateServer(int portNumber) {
	ServerSocket serverSocket = null;
	serverSocket = new ServerSocket(portNumber);
	while (!serverSocket.isClosed()) {
	Socket incomingClient = serverSock et.accept();
	processClientConnection(incoming Client);}
	return serverSocket.close().getStat us();}
		public ServerPort initializeListener(int listenPort) {
		ServerSocket listenerSocket = null;
		listenerSocket = new ServerSocket (listenPort);
		while (!listenerSocket.isClosed()) {
		Socket externalConnection = listen erSocket.accept();
		processIncomingRequest(externalC onnection);}
		return listenerSocket.bind(null).get LocalPort();}
			public void sockPipe(ServerSocket serverSocket, Socket cS, String m) {
			Socket cS = serverSocket.accept();
			BufferedReader in = new BufferedR eader(new InputStreamReader(clie ntSocket.getInputStream()));
			PrintWriter out = new PrintWriter(cl ientSocket.getOutputStream(), true);
			while ((m = in.readLine()) != null) { out.println(m); }
			clientSocket.close();
			serverSocket.close();}
				public static void sendInfo(Socket s ocket) {
				try {
				PrintWriter out = new PrintWriter(s ocket.getOutputStream(), true);
				BufferedReader in = new BufferedR eader(new InputStreamReader(sock et.getInputStream()));
				out.println();
				socket.close();
				} catch (IOException e) {
				e.printStackTrace();}}

补全结果	close().getStatus();	bind(null).getLocalPort();	closeStream(in);	close();
不使用检索增强	close().getStatus();			
真实值	close().getStatus();			

图 2 不同的检索算法对检索结果的影响结果案例

同时，通过实验我们发现对于同样的代码输入，不同的算法检索到的结果也不一定相同，产生的检索增强效果也不一定相同. 例如对于图 2 中的输入代码，待补全区域是最后一行的返回值语句，需要返回一个 ServerStatus 类型的对象. 四种检索算法得到的检索结果各不相同，其中 BM25 算法检索得到的代码的返回值类型与输入代码的返回值类型一致，TF-IDF 算法检索得到的代码则在实现功能和返回值类型上与输入代码都不一致；两种向量检索方法则检索出了不包含返回值的代码，与原输入代码的待补全位置需要返回 ServerStatus 对象的功能不符. 相应地，使用了 BM25 算法进行检索增强的实验组得到了正确的代码补全结果，而使用其它三种算法进行检索增强的实验组则导致模型产生了错误的输出. 此外，我们从实验样例中也观察到，对于类似图 2 的情况，即使不使用检索增强方法，生成模型也能够产生正确的代码补全结果. 这说明模型并非在任何情况下都需要依赖检索增强的提示信息来保证代码补全效果，低质量的检索结果反而可能会误导模型.

启示 2：据此，我们推论出通过改进代码检索方法，提升检索得到的代码在功能、结构上与输入代码的相关性，并在检索后根据代码检索结果灵活决策是否使用检索增强方法，可能能够在一定程度上提升检索增强的效果.

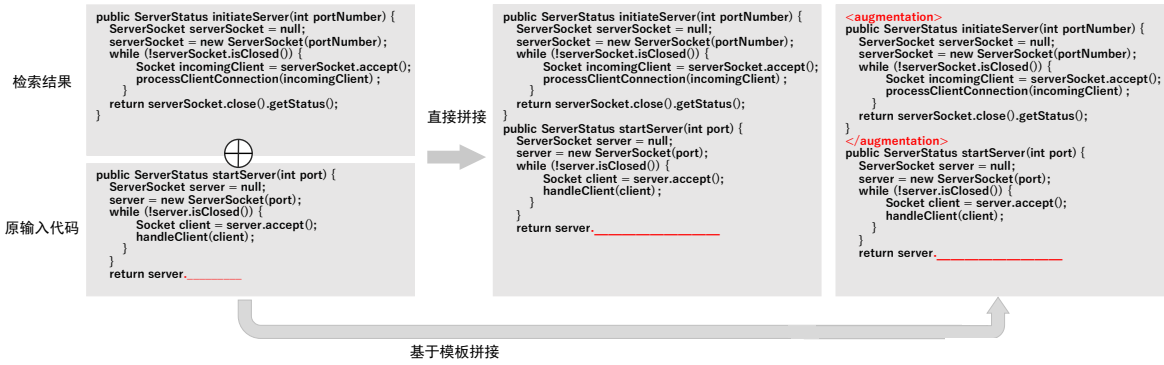

图 3 直接拼接与基于模板拼接对比

2.3 代码检索结果的后处理方法对基于检索增强的代码补全技术的影响（因素c）

现有的基于检索增强的代码补全方法在检索到代码片段后，有两种结合检索结果与输入代码的方法：一种是直接将检索到的代码片段与输入代码用字符串拼接的方式拼接在一起[9]，另一种是将检索后的代码加入到特定的提示模板，再与输入代码拼接到一起[14].

表 3 直接拼接和基于模板拼接后的检索增强效果对比

评估指标	直接拼接			基于模板拼接		
	94K	346K	9680K	94K	346K	9680K
ES	63.29	64.86	65.82	63.48	65.20	**66.15**
EM	36.62	37.32	39.56	36.54	37.71	**39.79**

为了探索两种检索结果的处理方式对于代码补全效果的影响，我们设计了一组对比试验，分别利用两种代码拼接的方式进行检索增强. 实验设计的代码拼接方法如图 3 所示，图中的红色下划线代表待补全的代码位置，并不包括在实际实验的输入中. 对于基于模板的拼接方式，我们将一对名为$< augmentation >$的分隔标签加入到输入代码和检索结果之间，用于区分检索得到的代码和原始的输入代码. 我们尝试在由 9.4 万、34.6 万、968 万条代码构成的三种数据规模的方法级代码检索库上，在 CodeXGLUE 数据集的测试集代码上先使用 BM25 算法检索出代码，然后分别使用直接拼接和基于模板的拼接两种结合方式进行检索增强，测试代码补全效果. 最终得到的实验结果如表 3 所示.

从表中可以看出，在检索算法固定不变的情况下，加入简单的分隔标签作为提示模板来结合检索结果与输入代码后，在由数据规模为 34.6 万和 968 万条代码构建出的代码检索库上，使用简单的提示模板进行代码拼接的检索增强方法在两项评估指标上均高于直接进行字符串拼接的检索增强方法. 其中，在由 968 万条代码构建出的代码检索库上，基于提示模板进行代码拼接实现的检索增强方法产生的代码补全效果在两项指标上要比使用直接的代码拼接实现的效果分别高出 0.5 个百分点和 0.6 个百分点.

启示 3：据此，我们推论出通过改进对代码检索结果的后处理方法，通过有效的提示模板将检索结果与输入代码结合起来，可能能够提升基于检索增强的代码补全效果.

综上，我们在经验研究中重点识别了检索库的代码粒度、代码检索方法、检索结果的后处理方法这三个影响检索增强效果的因素，并通过半定性半定量实验对应得出了三点可能能够优化检索增强效果的启示：

第一，通过扩大检索库规模并选择更精细的代码表征粒度，可能能够提升检索增强效果. 根据针对因素 a 的分析，基于 968 万条代码构建的检索库进行检索增强的效果，远好于在 9.4 万和 34.6 万条代码上构建检索库的检索增强效果. 检索库的规模越大，检索库中代码的多样性越强，从而更有可能检索到与输入代码具有相似结构或语义的提示信息. 同时，实验表明方法级代码检索库的检索增强效果好于文件级代码检索库. 由于方法体是从完整代码体中更精细划分出的子模块，因而这启发我们选择更精细的代码表征粒度可能也对检索增强效果有提升作用，例如对方法体进行进一步的切分，扩充代码检索库的多样性.

第二，通过改进代码检索方法，提升代码检索结果在功能、结构上与输入代码的相关性，并在检索后根据代码检索结果的质量灵活决策是否使用检索增强方法，可能能够提升检索增强的效果. 根据针对因素 b 的分析，使用 BM25 算法搜索文本类型代码的检索增强效果要好于使用 TF-IDF 算法和向量检索算法的效果，这可能是因为 BM25 算法能够综合考虑检索代码的词频和文本长度，而向量检索的方式虽然能够以相似性度量代码间的语义关系，但无法确保检索结果中包含代码待补全位置所需要的下文信息，例如图 2 中的两种向量检索算法虽然检索到了与输入代码类似的 Socket 通信代码，但代码结果中并不包含方法的返回值，与原输入代码的待补全位置需要返回 ServerStatus 对象的功能不符. 因而这启发我们可能可以将 BM25 算法应用于代码检索，将向量相似性计算应用于衡量检索结果的质量.

第三，通过有效的提示模板将检索结果与输入代码结合起来，可能能够提升检索增强的效果. 根据针对因素 c 的分析，使用模板将检索结果与原始输入代码区分开之后，代码补全的效果要比直接将检索结果与原始输入代码拼接的方式更好，这可能说明直接拼接的方式会导致模型无法区分输入内容的哪一部分是提示信

息、哪一部分是真实输入信息，从而影响模型的推理. 这启发我们利用提示模板对生成模型进行微调，从而提升检索增强的效果.

3 基于多阶段改进检索增强的代码补全方法

软件开发人员在编写代码时，常常会参考项目中具有相似语义的代码片段. 代码生成模型在根据检索增强的结果生成输出代码片段时也具有类似的特点，会以输入中提供的代码上下文作为参考[15]. 本文从第 2 章经验研究得出的三种因素入手，提出了一种基于多阶段改进检索增强的代码补全方法 MAGIC（Multi-stAGe optImization for retrieval augmentated Code completion），可以提升代码检索模块对代码生成模型的辅助生成作用，并减少模型在代码生成阶段受到的无关代码干扰，提升生成代码的质量.

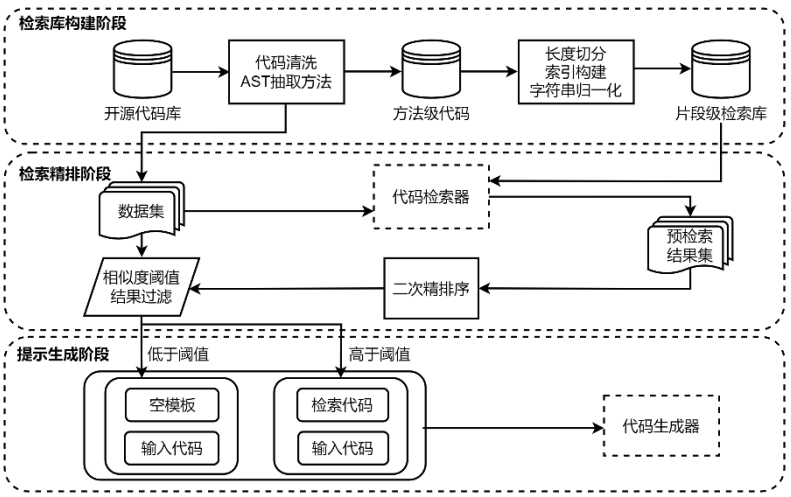

图 4 基于经验研究的检索增强改进方法流程图

3.1 方法概述

本文基于经验研究得出的结论，从代码补全中的检索增强的三个主要阶段：检索预处理阶段、检索阶段、检索后生成前阶段分别进行改进优化. 本文设计的方法如图 4 所示. 针对检索增强代码补全方法的特点，本文的改进方法主要分为三个阶段：检索库构建阶段（对应经验研究因素 a）、检索精排阶段（对应经验研究因素 b）、提示生成阶段（对应经验研究因素 c）.

- **检索库构建阶段**. 这一阶段针对经验研究的因素 a 进行改进. 由于经验研究发现方法级别的代码库比文件级别的代码库效果更好，这启发我们使用更细粒度的代码库提升检索增强效果，所以本文在方法级别代码库的基础上再做代码切分策略，将代码库划分为更小的粒度，扩充代码检索库的规模并提升代码内容的多样性. 因此在检索库构建阶段，我们主要基于抽象语法树和代码切分规则构建高质量的片段级代码检索库. 首先，对开源代码库进行清洗，筛除不包含有效方法片段（例如单元测试代码、不包括方法实现的代码）、或无法解析为抽象语法树的代码. 接着，基于抽象语法树从代码中抽取出方法体，将方法中的用户自定义变量名、常量归一化为统一的特殊标识；并根据方法级代码的 token 数量的分布情况确定代码片段切分长度. 最后，将方法级代码切分为指定长度范围内的多个代码片段，并标记同一段代码切分出的子代码片段之间的顺序关系，存储在 ElasticSearch 数据库中.
- **检索精排阶段**. 这一阶段针对经验研究的因素 b 进行改进. 在检索精排阶段，主要利用排序算法获得与当前输入代码最匹配的代码片段. 由上文的经验研究可知，基于 BM25 算法进行检索增强的效果要

优于 TF-IDF 算法和向量检索算法，因而我们在 BM25 检索方法的基础上进行优化. 首先，基于字符串级别的 BM25 搜索算法，从检索库中检索出前k个候选代码片段. 接着，根据检索的 BM25 分数对候选项进行粗排序，并记录每个选项对应的排名. 最后基于字符串的 Ratcliff 结构相似度和深度学习模型计算的向量相似度的融合分数进行二次精排序，取得与输入代码匹配度最高的检索结果.

- **提示生成阶段**. 这一阶段针对经验研究的因素 c 进行改进. 在提示生成阶段，根据检索到的代码片段构建提示模板，利用集成了提示模板的代码数据对模型做微调训练，提升生成模型的代码补全效果. 根据经验研究对因素 c 的分析，利用合适的特殊符号将检索结果与输入代码分隔开有可能能够提升代码补全效果. 因此我们利用训练好的模型计算出输入代码与检索结果代码的相似度，根据相似度和实际人工观测的代码语义相似情况设置过滤阈值，并将相似度高于阈值的检索结果代码添加到我们预先设计的提示模板中，与输入代码拼接后输入给生成模型；将相似度低于阈值的检索结果筛去，仅添加空的提示模板与输入代码拼接后，输入给生成模型.

3.2 检索库构建阶段（因素a）

本文所构建的片段级代码库，是在方法级代码库的基础上进行代码切分得到的，相比于经验研究因素 a 实验中使用的方法级代码库和文件级代码库，具有更小的代码粒度. 在构建片段级代码检索库时，考虑到我们的方法在工业界的实际可用性和易用性，我们选择使用 ElasticSearch 作为检索库的存储数据库，因为 ElasticSearch 使用倒排索引和基于 BM25 算法的检索技术，能够快速地在大规模数据集中进行字符串级别的检索，并针对每一个搜索结果计算得到 BM25 分数，能够帮助我们筛选结构相似度高的代码片段；对于大规模的代码库来说，ElasticSearch 采用分布式架构，可以将索引的代码片段数据划分到多个分片上，分布在不同的节点上进行并行处理，提高了处理能力和可伸缩性，适合本文的在大规模检索库上进行代码检索的场景.

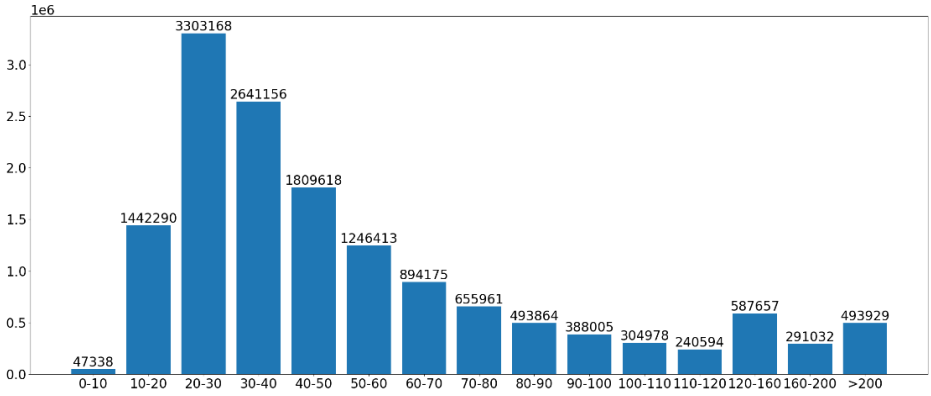

图 5 方法级代码的长度分布情况

首先，我们基于抽象语法树从开源代码库中抽取出了 1480 万个 Java 方法的代码. 为了在开源代码有限的情况下进一步扩充检索库的规模，从而提升检索库中代码的多样性、提升代码检索的命中率，我们采用按照指定长度进行代码切分的方式，处理抽取出的方法级代码. 具体包括如下两个步骤：

1. **统计代码长度分布情况**. 在通过切分代码的方式扩充检索库的规模时，需要确定合适的代码切分长度. 根据经验研究的启示 1，合理的代码粒度可能会对最终的补全效果有提升作用，而在 BM25 算法中，较长的文档可能会因为包含更多不相关的词项而得到较低的相关性得分. 因此我们进一步统计代码长度分布情况，并结合 BM25 算法的特点来确定合理的切分长度. 我们定义方法代码的长度为该方法所包含的按照空格划分的 token 的数量，统计了 1480 万条 Java 方法按照空格进行分词之后的序列长度，方法级代码的长度分布情况如图 5 所示.
2. **切分代码片段并构建顺序索引**. 从图 5 中可知，有 5,944,324 个方法的 token 数量在[20, 40)的区间内，总数

在所有长度区间中排名最高. 我们以代码长度分布区间为参考，确定代码切分长度L. 按照长度L切分代码后，所有按照空格分割后产生的 token 数量大于L的代码都将被分割为至少 2 个子片段，从而有效扩充检索库规模. 在切分代码时，我们会维护一个记录代码段之间的上下文关系的顺序索引. 顺序索引记录了切分前各个子片段之间的上下文依赖关系，它的作用是使检索器在检索代码时能够快速根据当前匹配到的代码片段找到它对应的上下文，从而在切分后的代码检索库中有效保留原有的方法级代码库的完整信息，避免代码切分导致的检索库代码顺序混乱问题. 代码切分算法的主要过程如下：

(a) 维护一个全局变量$index$，用于记录切分后的代码片段的id. 将$index$的初始值设置为 0.

(b) 维护一个全局索引列表$book$，用于记录切分后的代码片段之间的上下文关系. $book[index]$的值代表了id为$index$的代码片段的下文代码片段的id. 如果$book[index]$的值与$index$的值相等，则说明当前的代码片段是其所属的方法片段切分出来的最后一段代码，不包含对应的下文.

(c) 顺序地遍历代码库的所有代码. 对于每个完整代码，每间隔L个 token 就对其切分一次，并在检索库中将该切分出的代码片段的id标识为$index$. 若当前代码包含的 token 的数量大于L，说明当前的子代码片段还存在下文，则将$book[index]$的值设置为$index+1$；否则说明已经切分到当前方法代码的末尾，则将剩余的包含 token 数不足L的代码片段直接保存在检索库中，并将$book[index]$的值设置为$index$，代表本轮方法代码切分完毕.

(d) 设置$index = index + 1$，继续遍历代码库的下一段方法代码，直到对所有的代码都完成步骤(c)的操作.

上述方法中，代码切分长度L由实际统计出的代码长度分布情况确定，我们在第 4 章实验中说明了代码切分长度L的取值. 此外，我们维护了一个记录代码片段之间顺序关系的索引列表，用于在检索库中保存代码片段之间的上下文关系信息.

图 6 给出了上述片段级代码切分算法按照指定长度切分代码并构建索引的方法示例. 图中的蓝色部分是切分前的方法级代码，绿色部分是切分后的片段级代码，灰色部分是记录代码片段的上下文关系顺序的索引. 图中黑色的箭头代表子代码片段之间的上下文关系，箭头指向每个子代码片段的下一个代码片段；红色的箭头代表该子代码片段是最后一段代码片段，因而该段代码的下文就是该段代码本身，所以红色箭头实际上指向的是该子代码片段自身的索引. 通过这种方式，将每个子片段都通过索引与其原有上下文关联起来，从而确保检索到子片段后能够快速获得与其相邻的其它子片段，补充检索结果的上下文信息.

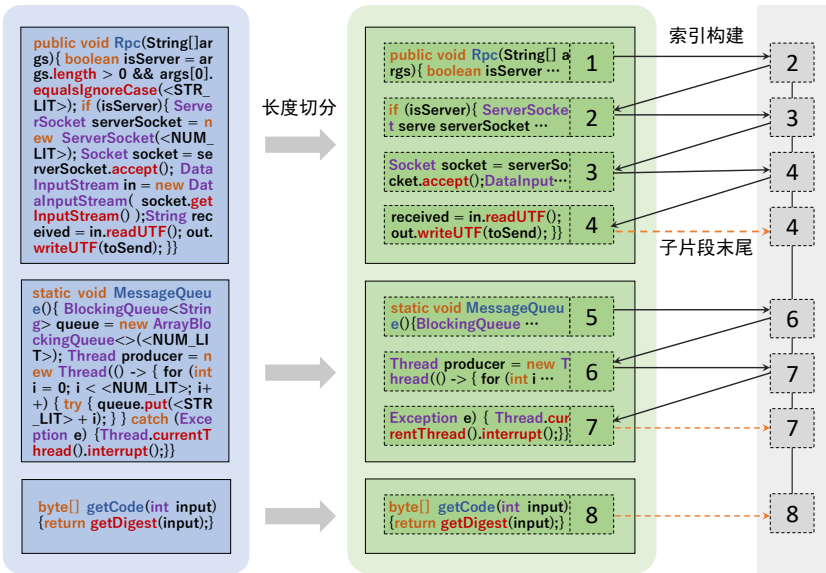

图 6 片段级代码切分与索引构建规则示意图

3.3 检索精排阶段（因素b）

根据经验研究启示 2，提升检索得到的代码在功能、结构上与输入代码的相关性，并在根据代码检索结果灵活决策是否使用检索增强方法，可能能够在一定程度上提升检索增强的效果. 同时，当向量检索库的规模达到百万级别以上时，向量类型代码检索的速度会大大下降. 在包含 1480 万条代码向量的向量类型检索库上批量检索 40,027 条查询向量，其单个代码向量的平均检索时间开销达到了 0.586 秒，而使用 BM25 算法进行字符串级别的检索的单条代码片段的平均检索时间开销仅有 0.039 秒. 据此，我们选择以 BM25 算法作为基础检索算法，将 BM25 算法应用于初步检索与粗排序，将代码向量相似度计算应用于小规模代码量的二次精排序，在避免大量检索时间开销的同时提升检索结果的准确度.

检索精排阶段针对经验研究所述因素 b 的改进策略主要包括基于倒秩融合策略进行二次排序、基于索引的代码片段拼接、基于相似度阈值过滤检索结果三个部分. 首先，从代码检索库中检索出一部分相关性分数最高的代码片段作为候选代码；然后使用深度学习模型输出的向量计算查询代码与候选代码的语义相似度，并使用字符串最长公共子序列算法计算查询代码与候选代码的结构相似度；接着，使用倒秩融合算法合并两种相似度计算方式的排序结果，并取倒秩融合分数排名最高的候选项，基于该候选代码片段对应的下文索引拼接出完整的代码下文；最后利用深度学习模型计算查询代码的向量与拼接后的代码块的向量之间的语义相似度，保留语义相似度高于预期阈值的结果. 本文提出的代码检索精排方法的整体流程如图 7 所示.

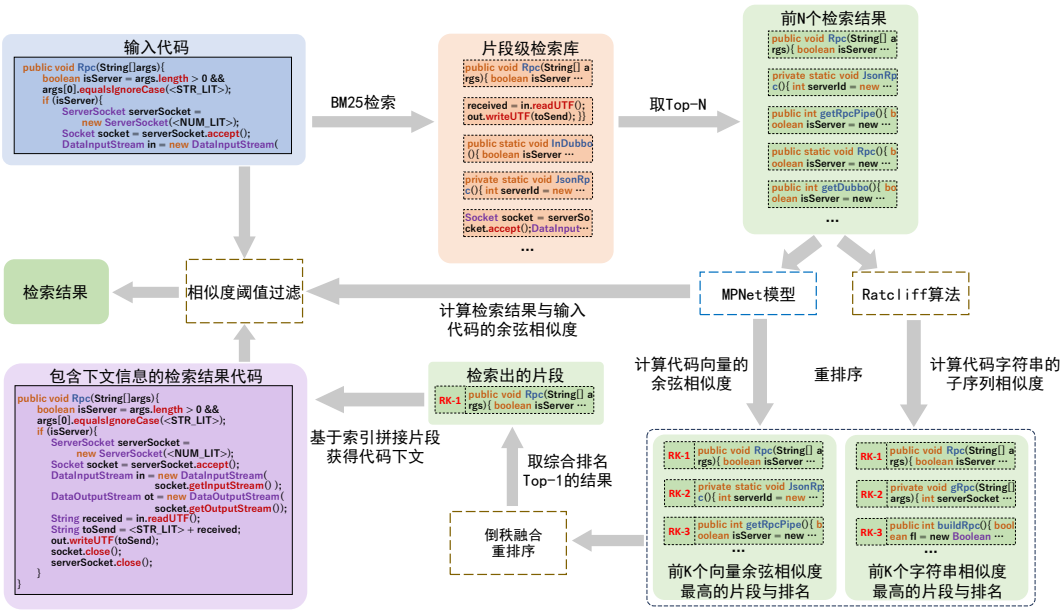

图 7 二次精排流程图

3.3.1 基于倒秩融合进行二次排序

本文设计改进的检索排序方法的具体步骤如下：

(a) **基于 BM25 算法检索出相关度最高的前N个候选代码片段D_b**. 这一步的作用是缩小可能的候选项范围，降低下一步排序的时间开销.

(b) **基于深度学习嵌入模型筛选出余弦相似度最高的前K个候选代码片段D_m**. 深度学习模型能够有效保留代码的语义信息. 因此我们利用预训练的嵌入模型将查询代码Q_c与候选代码片段D_m转化为稠密向量的形式，并计算两个向量之间的余弦相似度，然后依据余弦相似度对 BM25 算法的检索结果D_b的候选项进一步排序，保留前K个候选代码片段，同时记录其对应的分数排名. 其中，为确定具体的嵌入模型，我们尝试对比了 CodeBERT[30]、CodeT5[31]、MPNet[32]、CodeGPT[33]四种模型作为嵌入

模型的检索增强效果，实验发现选用以语义相似度为训练目标、以 CodeSearchNet 代码数据集等语料训练的 MPNet 模型得到的检索增强效果最好，在编辑相似度和完全匹配指标上比 CodeBERT 分别高出 0.72 个百分点和 1.30 个百分点，比 CodeT5 分别高出 0.78 个百分点和 0.55 个百分点，比 CodeGPT 分别高出 2.04 个百分点和 2.75 个百分点. 故而我们最终选择以 MPNet 模型作为本算法的嵌入模型.

(c) **基于 Ratcliff 算法筛选出字符串结构相似度最高的前K个候选代码片段D_r**. Ratcliff 算法能够输出两段代码之间的结构相似度. 我们利用 Ratcliff 算法计算出查询代码Q_c与D_b中每个代码片段的结构相似度并排序，保留前K个候选代码片段，同时记录其对应的分数排名. 在实际实验中，我们基于对检索时间长度、实际检索效果的综合考虑，取N的值为 100，取K的值为 50.

(d) **基于倒秩融合算法合并排序结果**. 我们利用倒秩融合算法，将 Ratcliff 算法输出的排序结果与代码的余弦相似度排序结果合并起来，得到新的排序结果，并取排名 Top-1 的代码片段作为本次检索的结果. 倒秩融合能够根据不同排名列表的可靠性和权重等因素，对结果进行更精细的调整和优化，同时避免了直接使用 BM25 分数与余弦相似度进行分数加权带来的权重比例难以合理分配的问题. 倒秩融合的计算公式如下：

$$Score(q) = \frac{1}{Rank_T(q) + M} + \frac{1}{Rank_B(q) + M} \tag{1}$$

其中，M代表的是平滑因子，一般取 60 或 61. $Rank_T(q)$和$Rank_B(q)$分别代表两种不同的相似度计算方法产生的候选项排名. 在本文方法中，$Rank_T(q)$是我们检索到的每个代码片段在其向量余弦相似度分数中的排名，$Rank_B(q)$是我们检索到的每个代码片段在其 Ratcliff 相似度分数中的排名. 我们基于倒秩融合算法计算出的分数，对候选项做二次精排序.

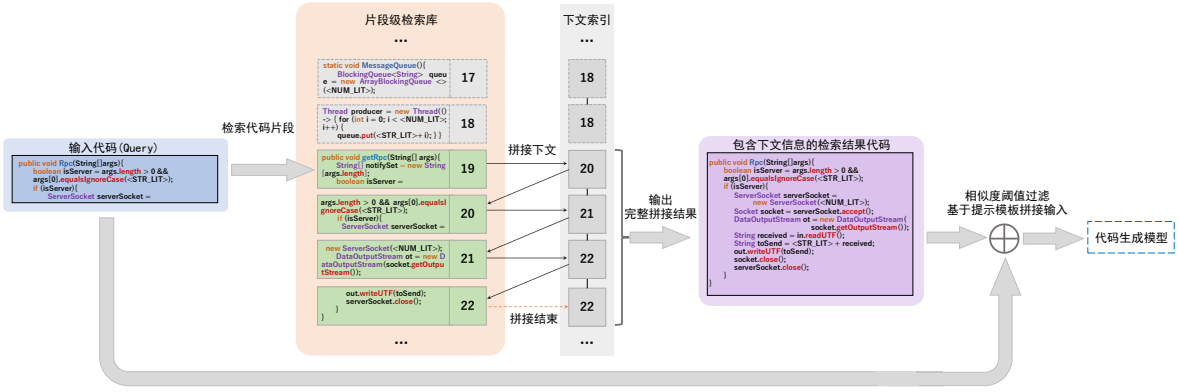

图 8 改进的检索增强代码拼接过程图

3.3.2 基于索引的代码片段拼接

检索增强的目的是为模型提供包含额外的上下文信息的检索结果. 本文方法的检索精排阶段获取到的检索结果是与输入代码片段相似度最高的代码片段，并不是完整的方法级代码体. 为了使得检索到的片段级结果中包含更多的下文信息，我们基于索引实现代码片段的上下文代码拼接,从而补全检索结果中的下文信息. 基于索引片段拼接获取下文代码的过程如图 8 所示. 具体方法流程如下：

(a) 读取在检索库构建阶段预先存储的记录代码上下文顺序的全局索引列表$book$，并初始化迭代变量$index = id$. 其中，$index$代表当前循环中遍历到的代码片段的id. 我们根据$index$获得$book[index]$的值.

(b) 如果当前$book[index]$的值不等于迭代变量$index$，说明当前代码片段还存在下文信息. 此时将$index$

设置为$book[index]$，重复步骤(a).

(c) 如果当前$book[index]$等于当前的迭代变量$index$，说明当前代码片段没有下文信息需要继续拼接. 结束流程.

需要注意的是，在根据预先构建的代码片段索引进行代码拼接操作时，我们的算法仅拼接当前匹配到的代码片段及其后续的有索引连接的对应代码片段，即仅拼接当前片段对应的下文代码，而不拼接当前片段对应的上文代码. 这主要有两方面考虑：一方面，我们考虑到代码生成模型本身的输入长度限制，在进行检索增强信息的拼接时，拼接多余的上文代码片段可能会导致输入结果更有可能超出模型所能接受的最大长度；另一方面，我们在实验中枚举了包括上文拼接和下文拼接在内的多种不同的代码拼接方式，当前设计的基于索引的下文片段拼接方法得到的代码生成效果在评测指标上的表现最好，我们将在实验部分展示这一点.

3.3.3 基于相似度阈值过滤检索结果

根据经验研究中对于因素 b 的分析，检索增强的结果代码对于生成模型来说并不一定可用，可能成为输入噪声而非提示信息，从而干扰模型的正常生成结果. 因此本文采用代码语义相似度评估的方式确定了一个代码过滤阈值S，在将检索结果代码与输入代码拼接之前，先计算检索结果代码与输入代码的语义相似度. 如果语义相似度大于预先设置的阈值S，则将检索结果代码与输入代码拼接起来作为提示信息输入模型；而如果语义相似度小于阈值，则认为检索结果代码是无关代码，此时将输入代码直接输入给模型，仅基于模型自身的能力去生成代码.

为了确定模型在何时采用自身能力生成、何时采用检索增强的相似度阈值进行过滤，需要确定阈值S的具体取值. 我们从数据集中抽取出共 40,027 条代码组成验证集，根据验证集的统计结果确定阈值. 首先针对验证集的每一条代码做预检索，取得每一条代码对应的检索结果，然后使用 MPNet 模型将两段代码转化为向量形式，计算两个向量之间的余弦相似度，得到的相似度分数统计结果如表 4 所示. 其中，我们使用的 MPNet 模型是 HuggingFace 开源的、在 CodeSearchNet 代码数据集上经过训练的 MPNet-base 语义嵌入模型，实验表明该模型在语义相似度计算和语义搜索任务上表现最好[34]. 给定输入代码，它会输出一个捕获语义信息的向量，可用于计算代码间的相似性任务.

表 4 验证集输入代码与检索结果的相似度统计及对应生成策略

相似度区间	代码数	累计区间	累计代码数	合并区间	代码数比例	模型生成策略
[0 , 0.1]	212	[0 , 0.1]	212			
(0.1, 0.2]	1,156	[0 , 0.2]	1,368	[0 , 0.4]	9,744(24.15%)	模型直接生成
(0.2, 0.3]	3,527	[0 , 0.3]	4,895			
(0.3, 0.4]	4,849	[0 , 0.4]	9,744			
(0.4, 0.5]	5,829	[0 , 0.5]	15,573			
(0.5, 0.6]	5,537	[0 , 0.6]	21,110	(0.4, 0.7]	15,591(39.75%)	
(0.6, 0.7]	4,225	[0 , 0.7]	25,335			检索增强生成
(0.7, 0.8]	4,314	[0 , 0.8]	29,649			
(0.8, 0.9]	4,883	[0 , 0.9]	34,532	(0.7, 1.0]	14,692(36.10%)	
(0.9, 1.0]	5,495	[0 , 1.0]	40,027			

从表 4 中可以看出，在验证集中有 24.15%的代码检索结果与其输入代码之间的语义相似度低于 0.4，有 39.75%的代码检索结果与其输入代码之间的语义相似度在 0.4 到 0.7 之间；有 36.10%的代码检索结果与其输入代码之间的语义相似度在 0.7 到 1.0 之间. 我们通过采样观察，将验证集的代码按照其与对应检索结果间的相似度大小划分为三种类型，三种类型的输入代码与其对应的检索结果如图 9 所示，图中标注的黄色部分为输入代码与检索结果的公共部分.

(a) **输入代码与检索结果的语义相似度较低**. 这部分代码与其对应检索结果之间的相似度值小于等于 0.4，输入代码与检索结果实现的功能逻辑不相关，具体的样例如图 9 第一对代码所示.

(b) **输入代码与检索结果的语义相似度中等，检索结果中包含部分模型所需的下文信息**. 这部分代码与其对应的检索结果之间的相似度大于 0.4 且小于 0.7，这部分输入代码对应的检索结果中会包含一些与输入代码相关的下文信息，但输入代码与检索结果的最长公共子序列长度小于等于字符串总长度的一半. 具体的样例如图 9 第二对代码所示.

(c) **输入代码与检索结果的语义相似度高，检索结果与输入代码的语义基本一致**. 这部分是与检索结果之间的相似度大于等于 0.7 的代码，这部分代码的功能实现与上下文信息与输入代码片段基本一致，且输入代码与检索结果的最长公共子序列长度大于字符串总长度的一半. 具体的样例如图 9 第三对代码所示.

输入代码	检索结果
输入代码与检索结果的语义不相关 [0, 0.4]	

public static void main(String[] args) throws Exception {
HttpClient httpClient = HttpClients.createDefault();
HttpGet httpGet = new HttpGet("https://api.com/data");
HttpResponse response = httpClient.execute(httpGet);
int statusCode = response.getStatusLine().
	public int getStatusRes() {
	int[] status = {1, 2, 3, 4, 5};
	int sum = Arrays.stream(status).sum();
	int average = sum / status.length;
	return average;
	}

检索结果中包含部分模型所需的下文信息 (0.4, 0.7]	

public boolean equals(Object obj){
if (getType() == IRuntimeLoadpathEntry.CONTAINER){
String id = getPath().segment(<NUM_LIT>);
LoadpathContainerInitializer initializer = RubyCore.getLoadpathContainerInitializer(id);
IRubyProject javaProject1 = getRubyProject();
Object comparisonID1 = initializer.
	if (getType() == IRuntimeLoadpathEntry.CONTAINER){
	Object comparisonID1 = initializer.getComparisonID(getPath(), javaProject1);
	Object comparisonID2 = initializer.getComparisonID(r.getPath(), javaProject2);
	return comparisonID1.equals(comparisonID2);
	}
	}

输入代码与检索结果的语义基本一致 (0.7, 1.0]	

public Component inflate(InputStream xml) throws LayoutInflaterException{
LayoutInflaterContentHandler handler = new LayoutInflaterContentHandler(componentManager);
XMLReader parser = XMLReaderFactory.createXMLReader();
parser.setContentHandler(handler);
parser.parse(new InputSource(xml));
return handler.
	public Component inflate(InputStream xml, List<ComponentProvider> componentProviders) {
	LayoutInflaterContentHandler contentHandler = new LayoutInflaterContentHandler(componentProviders);
	XMLReader parser = XMLReaderFactory.createXMLReader();
	parser.setContentHandler(contentHandler);
	parser.parse(new InputSource(xml));
	return contentHandler.root;
	}

图 9 相似度区间对应的代码样例

综上，根据从验证集中实验对比得到的输入代码与检索结果的相似度差异，我们将划分使用检索增强策略的阈值设置为 0.4 的相似度值，并将这一规则应用到测试集，用于后续的实验评估效果. 当输入代码与检索结果的语义相似度低于 0.4 时，我们直接将代码输入给模型，依靠模型自身能力生成结果；当语义相似度高于 0.4 时，我们将检索代码与输入代码拼接后再输入给模型，实现检索增强.

3.4 基于提示模板生成代码（因素c）

根据我们在第 2 章经验研究中关于因素 c 的分析，现有的检索增强代码生成框架会直接将检索得到的代码片段与输入代码片段拼接起来，但这样无法明确区分检索得到的外部知识与实际输入. 因此，我们设计了一个简单的提示模板用于区分检索得到的外部代码和原输入代码. 定义提示模板的目的是为了提供一个结构化的指导，以帮助模型生成准确、合理的输出. 我们的具体方法是，先在检索得到的代码片段的两侧封装上$<aug></aug>$标签，再将封装后的检索结果与原输入代码拼接起来，输入给代码生成模型. 如果当前检索结果与原输入代码之间的向量余弦相似度低于我们预设的阈值，则仅仅在原输入代码的前部添加一对空的$<aug></aug>$标签. 封装提示模板的结构如图 10 所示. 我们同样基于上述规则对代码生成模型进行模型微调. 需要注意的是，在经验研究中我们尝试使用了$<augmentation></augmentation>$的标签对来分隔输入代码与检索结果代码，但考虑到提示模板的简洁性，我们使用了更短的$<aug></aug>$标签. 在实验中发现，在不超过模型最大输入长度的情况下，利用这两种标签构建提示模板微调的模型的代码补全效果基本一致. 此外，我们将$<aug></aug>$标签加入到模型的特殊 token 列表中作为特殊提示符进行处理，从而避免分词阶段对提示模板进行拆分.

本文选用参数量为 124M 的 CodeGPT-adapted-java 预训练模型作为 MAGIC 方法中的生成模型. CodeGPT 是在 GPT-2 的基础上利用从 GitHub 等代码托管平台上获取的大量源代码训练的生成模型，它具有更好地理解

代码结构、功能和语法的能力[33].

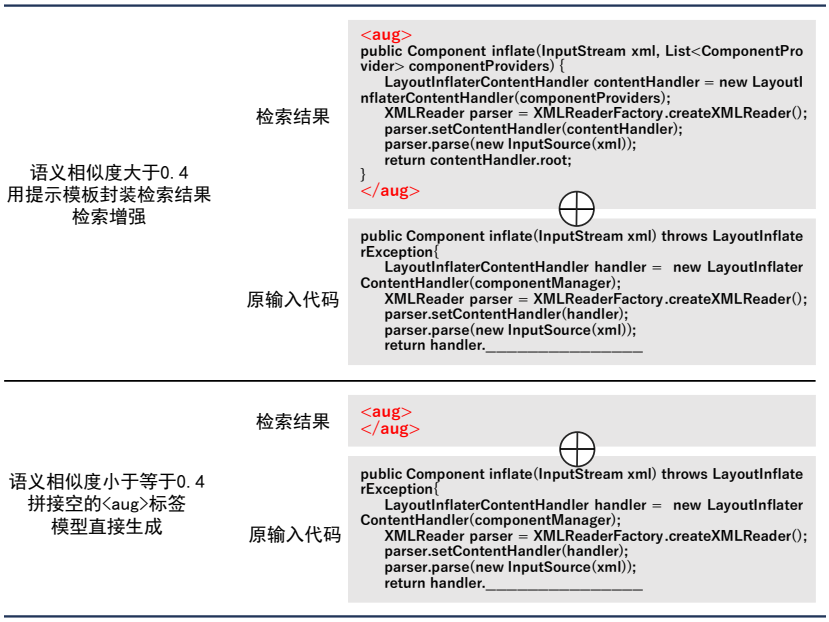

图 10 检索增强的提示模板拼接示意图

在训练模型的数据准备阶段，首先将训练集代码处理为提示模板的形式. 我们针对训练集在代码检索库中做一遍预检索，通过第 3.3 节设计的检索规则，检索得到训练集中每段代码对应的相关性最高的代码片段. 接着，计算输入代码和其对应检索结果之间的语义相似度，将相似度高于我们预先设置的阈值的检索结果代码封装$< aug ></aug >$标签，随后拼接到输入代码上；将相似度低于阈值的检索结果代码舍弃，仅拼接一对空的$< aug ></aug >$标签到输入代码上；最后将数据按照常规文本序列的输入方式，分批次输入给模型. 模型微调使用 Adam 优化器和交叉熵损失函数（Cross-Entropy Loss），设置权重衰减系数为 0.01，学习率为 8×10^{-5}，共微调训练 3 个 epoch.

4 实验评估

为了验证本文针对经验研究提出的多阶段改进检索增强的代码补全方法（Multi-stAGe optImization for retrieval augmentated Code completion, MAGIC）的有效性，实验部分主要采用了对照实验、消融实验、比较实验三种实验方式：通过对照实验，比较了现有的微软提出的 ReACC 检索增强代码补全框架与 MAGIC 方法在评估指标上的差距；通过消融实验，来验证 MAGIC 方法的三个主要优化策略的有效性；通过比较实验，来对比 MAGIC 方法与现有的参数量更大的模型在推理性能与资源开销方面的差距. 因此，我们将围绕以下四个问题展开实验验证：

- RQ1：本文提出的 MAGIC 方法是否优于现有方法？
- RQ2：根据经验研究中的三个关键因素所设计的代码片段切分、二次精排序、相似度阈值过滤、模板提示生成四种改进检索增强方法，是否能够提升模型的代码补全性能？
- RQ3：本文在检索库构建阶段构建的子代码片段索引和在检索精排阶段基于索引实现的代码上下文拼接策略，对检索增强代码补全的效果影响如何？
- RQ4：本文提出的方法与现有的参数量更大的生成模型相比，在推理性能与资源开销上的表现如何？

4.1 实验设置

4.1.1 环境配置

本文所有实验均在阿里巴巴集团企业级生产环境下完成. 硬件配置为具有 8 核 CPU、120G 内存和 4 块 24G 的 NVIDIA A10 显卡的阿里云 ECS 服务器.

4.1.2 模型与数据集

本文使用到的预训练模型主要包括一个生成模型和一个语义相似度计算模型. 本文以参数量为 124M 的 CodeGPT 作为生成模型，模型的输入是添加了提示模板的方法级 Java 代码，输出是根据上文补全出的代码片段；语义相似度计算模型则采用参数量为 110M 的 MPNet-base 模型，模型的输入是字符串级别的 Java 代码片段，模型的输出是表征该代码片段语义的高维向量，我们通过计算两段代码的高维向量之间的余弦相似度来衡量其语义相似度. 模型运行环境为 X86-64 位的 Linux 操作系统，使用 NVIDIA A10 显卡.

在数据集方面，本文使用微软开源的 GitHub Java Corpus 代码补全评估数据集[33]，该数据集已经包含了未完成的代码片段以及补全后的完整代码，可用于评估行级别代码补全任务. 我们从数据集中抽取出 94,678 个 Java 方法作为微调代码补全模型的训练集，抽取出 40,027 个 Java 方法作为测试集评估 MAGIC 方法的代码补全性能. 考虑到我们在下文中使用的 CodeGen 模型的训练数据来自 Google BigQuery，其中可能包含部分来自 GitHub 的代码，为避免测试数据泄露，本文在构建测试集时预先排除掉了测试集中与 BigQuery 数据集相同的代码；此外，为避免数据集内容与本文构建的代码检索库内容重叠，本文在实验前对大规模代码检索库和 GitHub Java Corpus 数据集进行了去重处理，排除掉了一模一样的代码，确保不出现数据重叠的问题.

4.1.3 代码检索库

我们爬取了 GitHub 平台共 45,196 个开发者的 145,532 个代码仓库的 Java 代码，并经过代码解析抽取出 14,840,178 个 Java 方法，按照每 30 个 token 为一个子代码片段的规则进行切分，最终切分出 36,903,876 个代码片段. 我们将切分出的代码片段以字符串形式存储在 ElasticSearch 中，构建出片段级代码检索库. 本文设计的代码切分策略仅用于构建大规模代码检索库，其目的是在有限的源代码上扩充检索库规模并增加代码多样性，提升代码检索的效果；而本文使用的训练集和测试集仅用于模型的训练与评估，并不参与检索库的构建过程，所以不需要被切分为代码片段.

4.1.4 对比方法

本文提出的 MAGIC 方法主要与现有的基于大模型的代码补全模型和基于检索增强的代码补全模型进行了对比. 具体包括如下几种模型：

- **GPT-2**：GPT-2 是 OpenAI 开发的一个基于 Transformer 架构的语言模型[35]，它通过堆叠 Transformer 的解码器模块实现，在包括代码生成和代码补全领域在内的多种文本生成任务上的表现都非常出色.
- **CodeGPT**：CodeGPT 是在 GPT-2 的基础上利用从 GitHub 等代码托管平台上获取的大量源代码训练的生成模型，它相比于 GPT-2 具有更好地理解代码结构、功能和语法的能力. 本文主要使用 CodeGPT-adapted-java 模型作为基础模型，同时将我们微调后的 CodeGPT 模型作为 MAGIC 方法的生成模型.
- **CodeGen**：CodeGen 是一种用于程序合成的自回归模型[36]，该模型在包括 The Pile、BigQuery 和 BigPython 在内的各种数据集上进行训练得到，具有较强的编程语言理解能力. 本文使用了 20 亿、60 亿、160 亿这三种不同参数量的 CodeGen 模型，用于比较 MAGIC 方法与现有的大模型之间的差异.
- **ReACC**：ReACC 是在 CodeGPT 模型的基础上，利用向量与 BM25 混合检索增强技术优化后的代码补全方法，它旨在利用参数量较小的模型达到更优的代码补全性能.

4.2 代码补全效果对比（RQ1）

本文使用编辑相似度（Edit Similarity）和完全匹配（Exact Match）指标来评估模型.

- **编辑相似度**：编辑相似度用于度量模型生成内容与真实结果之间的相似程度，该指标通过计算两个序列之间的编辑距离来表示. 编辑距离表示通过插入、删除和替换操作将一个序列转换为另一个序列所

需的最小操作次数，可以通过动态规划算法求出. 设模型的输出代码为$pred$，实际真实结果为gt. 其编辑距离为$edit(pred, gt)$，那么编辑相似度被定义为：

$$Edit\ Similarity(pred, gt) = 1 - \frac{edit(pred, gt)}{\max(|pred|, |gt|)} \tag{2}$$

- **完全匹配**：完全匹配用于衡量模型生成内容是否与真实结果完全一致，即是否完全匹配. 如果模型生成的内容与真实结果完全一致，则当前完全匹配值为 1，否则为 0，将所有的匹配值累加后，该匹配数量占总数量的百分比即为总的完全匹配值. 假设测试集包含N条测试数据，第i条测试数据的输出结果为$pred_i$，真实结果为gt_i，$equal$是判断两段代码是否完全一致的函数. 那么完全匹配计算公式如下：

$$Exact\ Match = \frac{\sum_{i=1}^{N} equal(pred_i, gt_i)}{N} \tag{3}$$

其中，代表判断模型对单条测试用例的生成结果与真实结果是否完全一致的函数$equal$为：

$$equal(pred_i, gt_i) = \begin{cases} 1 & pred_i = gt_i \\ 0 & pred_i \neq gt_i \end{cases} \tag{4}$$

实验对比结果见表 5. 其中，由于 MAGIC 方法和 ReACC 方法都使用到了代码检索库进行代码检索，故而其单条代码片段推理时间除模型推理时间外，也计入了代码检索所需的时间；两种检索增强方法的内存占用量是模型文件和检索库的总内存占用量.

表 5 MAGIC 改进方法与其他方法的代码补全评估指标对比

模型	编辑相似度	完全匹配	单条推理时间(s)	显存占用(MiB)	内存占用(MiB)
GPT-2	61.29	27.25	0.64	1,457	548
CodeGPT	62.20	29.93	0.47	1,610	510
CodeGen-2B	65.91	39.95	3.20	8,642	5,826
CodeGen-6B	66.53	40.75	3.66	41,972	14,643
CodeGen-16B	67.13	43.35	5.34	66,464	32,768
ReACC	65.82	39.56	1.98	2,205	15,695
MAGIC	70.27	42.65	0.96	2,327	16,133

从表中可以看出：与 GPT-2 和 CodeGPT 这种参数量较小的生成模型相比，MAGIC 方法在编辑相似度、完全匹配指标上都有明显的提升. 相比于 GPT-2 分别提升了 14.65 个百分点和 56.51 个百分点；相比于 CodeGPT 提升了 12.97 个百分点和 42.49 个百分点. 由于我们使用了基于检索增强流程微调后的 CodeGPT 模型作为代码生成模型，因而这也证明我们的代码检索增强补全优化方法对于提升生成模型的代码补全性能是有效的.

与参数量更大的 CodeGen 模型相比，MAGIC 方法相比于 20 亿参数量的 CodeGen 模型在两个指标上分别提升了 6.62 个百分点和 6.76 个百分点；而相比于 60 亿参数量的 CodeGen 模型在两个指标上则分别提升了 5.62 个百分点和 4.66 个百分点. 考虑到 MAGIC 方法使用的 CodeGPT 生成模型只有 1.24 亿个参数，这说明本文的方法在代码补全效果上总体超过了参数量为本文模型 48 倍的代码补全模型.

与微软提出的 ReACC 检索增强代码补全方法相比，MAGIC 方法在两个指标上分别提升了 6.76 个百分点和 7.81 个百分点. 同时，从表中可知 MAGIC 方法相比于 ReACC 检索增强方法能够节约 52.6%的推理时间. 由于我们的方法与 ReACC 方法使用了相同来源与规模的代码检索库、相同的 CodeGPT 生成模型架构，这说明我们的优化方法整体要优于 ReACC 方法.

4.3 改进方法的有效性评估（RQ2）

为了探究我们提出的改进方法中每一种优化方式的有效性，我们针对代码片段切分、二次精排序、相似度阈值过滤、模板提示生成这四种优化方法设计了消融实验，证明每一个独立模块的优化效果.

在我们的四种优化方法中，代码片段切分属于在数据层面的操作，二次精排序、相似度阈值过滤、模板提示生成这三种方法属于在模型算法层面的操作. 因此我们设计了两组实验：

- **实验一：代码片段切分方法的有效性验证实验**. 在实验一中，我们在不做任何模型或算法层面的优化

的情况下，枚举所有可能的代码片段切分长度，按照不同的切分长度，分别构建片段级代码检索库. 这一实验用于探究最优的构建片段级代码检索库的切分长度，并验证切分后的代码检索库相比于未切分的代码库在提升检索增强代码补全能力上的有效性.

- **实验二：检索增强与生成方法的有效性验证实验**. 在实验二中，我们将二次精排序、相似度阈值过滤、模板提示生成这三种优化方式排列组合为三组不同的方法，并在未经过切分的代码检索库与经过切分过的片段级代码检索库上分别进行检索增强代码补全实验. 这一实验用于验证 MAGIC 方法每一模块的有效性.

表 6 初始代码库各段代码按空格分割产生的 token 数量统计

token 数量范围	范围内代码数量	范围内代码数占比	累计代码数量	累计比例
0 ≤ count ≤ 10	47,338	0.318%	47,338	0.318%
10 < count ≤ 20	1,442,290	9.719%	1,489,628	10.03%
20 < count ≤ 30	3,303,168	22.26%	4,792,796	32.29%
30 < count ≤ 40	2,641,156	17.79%	7,433,952	50.09%
40 < count ≤ 50	1,809,618	12.19%	9,243,570	62.28%
50 < count ≤ 60	1,246,413	8.398%	10,489,983	70.67%
60 < count ≤ 70	894,175	6.025%	11,384,158	76.70%
70 < count ≤ 80	655,961	4.420%	12,040,119	81.12%
80 < count ≤ 90	493,864	3.328%	12,533,983	84.45%
90 < count ≤ 100	388,005	2.614%	12,921,988	87.06%
100 < count ≤ 110	304,978	2.055%	13,226,966	89.12%
110 < count ≤ 120	240,594	1.621%	13,467,560	90.74%
120 < count ≤ 130	191,396	1.289%	13,658,956	92.03%
130 < count ≤ 140	158,509	1.068%	13,817,465	93.09%
140 < count ≤ 150	129,571	0.873%	13,947,036	93.91%
count > 150	893,142	6.018%	14,840,178	100.0%

4.3.1 实验一：代码片段切分方法的有效性验证实验

我们设计代码片段切分的目的是增加代码检索库中的代码多样性，从而在代码量有限的情况下扩充代码检索库的规模，从而间接提升检索增强的效果. 为了探究这种通过按照指定长度切分代码来扩充检索库规模的方式是否对检索增强生成的效果有提升，我们首先统计了整个方法级代码检索库中所有代码按照空格分割后产生的 token 的数量，得到数量分布情况如图 5 的柱状图以及表 6 的具体数据所示.

从表中可以看出，方法级代码按照空格分割出的 token 数量有 50.09%集中在[0,40]的区间范围内，有 87.06%的代码分割后的 token 数量小于等于 100，仅有 12.84%的方法级代码包含的 token 数量超过 100. 当分割代码的 token 数量设定不小于区间下限、不大于区间上限时，则会有对应比例的代码被切分为片段. 例如，如果我们设置每经过 30 个 token 就切分一次代码，那么会有 67.71%的代码都被切分为多段包含不多于 30 个 token 的代码片段，则切分代码后检索库的代码数量规模会扩大到 36,903,876 个代码片段，为原检索库代码数量的 2.49 倍.

由实验的统计结果可知，切分的 token 数量设置得越大，则由切分操作产生的新代码片段的数量就会越少. 因此我们以 10 个 token 为一个步长，设置代码切分的 token 数量下限为 0、token 数量上限为 100，枚举从 0 到 100 个 token 范围内的所有不同的代码切分长度，根据每种长度分别建立检索库，并且仅使用 BM25 算法实现检索增强、使用 CodeGPT 模型生成代码补全结果. 我们对比了不同实验配置下的代码补全指标，得到结果如表 7 所示.

表 7 中的第一行切分 token 数量为 0 指的是不做任何切分，直接使用原始完整的代码检索库进行检索增强；最后一行则代表累加所有切分后的代码结果，它指的是我们将原完整代码检索库和按照不同 token 数量的代码切分策略构建出的所有检索库合并起来，构建出的包含 3.38 亿条代码片段的大型检索库，其数据规模是原代码检索库的 22.83 倍.

表 7 基于不同 token 数量切分代码后的检索增强效果对比

切分 token 数量	切分后检索库规模	扩增倍数	编辑相似度	完全匹配
0	14,840,178	1.00	65.82	39.56
10	97,045,936	6.54	63.22	37.06
20	52,318,537	3.53	66.63	39.14
30	36,903,876	2.49	67.79	40.31
40	29,268,944	1.97	67.20	39.83
50	24,957,243	1.68	67.17	39.75
60	22,289,348	1.50	66.65	39.61
70	20,529,973	1.38	66.19	39.61
80	19,309,185	1.30	65.83	39.61
90	18,427,316	1.24	65.87	39.61
100	17,767,395	1.19	65.98	39.61
[0 , 100]	353,657,931	22.83	67.30	39.63

从表 7 中可以看出，在不优化检索方法与模型的情况下，通过切分更短的代码片段构建更大规模检索库的方式能够影响检索增强代码补全的效果. 当选择以每 30 个 token 的数量作为一个单位来切分更小粒度的代码片段时，检索增强代码补全的效果最好，相比于未进行代码切分的实验组在两个指标上分别提升了 2.99 个百分点和 1.90 个百分点. 这证明了我们设计的代码切分方法的有效性.

本实验也证明代码检索库并非切分得规模越大就越好. 例如当以每 10 个 token 为一个单位切分所有代码时，尽管切分后的检索库规模达到了 97,045,936 个代码片段，扩增了 6.54 倍，但检索增强代码补全的效果还没有直接使用原始代码库的效果更好. 此外，我们将所有切分后的片段级检索库合并在一起构建 3.53 亿规模的超大型代码检索库后，检索增强代码补全的效果相比于各个子检索库的实验并没有较大的提升，这也说明在初始代码库代码数量有限的情况下，通过切分代码扩充检索库规模起到的优化效果并不会随着切分次数的增多而无限制地提升.

表 8 三种优化方法的消融实验结果

优化方法	编辑相似度	完全匹配
无优化	65.82	39.56
相似过滤	67.29	39.83
二次精排	69.70	42.00
提示生成	69.81	42.12
二次精排+相似过滤	70.06	42.57
二次精排+提示生成	70.11	42.50
提示生成+相似过滤	70.09	42.37
二次精排+相似过滤+提示生成	70.27	42.65

4.3.2 实验二：检索增强与生成方法的有效性验证实验

为了验证本文针对经验研究提出的三个因素所设计的二次精排序、相似度阈值过滤、模板提示生成三种优化方式对检索增强代码补全的有效性，我们基于实验一中检索增强效果最好的代码检索库（即按照每 30 个 token 的数量切分后的片段级检索库）进行检索增强代码补全的消融实验，测试了三种优化方法的独立作用效果，并将这三种优化方法两两组合进行性能指标结果对比，从而验证其提升代码补全效果的有效性. 实验结果如表 8 所示.

从表中可以看出，单独使用相似度阈值过滤方法能够将代码补全的两项指标分别提升 2.23 个百分点和 0.68 个百分点，单独使用二次精排序方法能够将代码补全的两项指标分别提升 5.89 个百分点和 6.17 个百分点，单独使用提示模板生成方法能够将代码补全的两项指标分别提升 6.06 个百分点和 6.47 个百分点；且任意将三种优化方法两两组合都能得到比单独使用一种优化方法更好的代码补全效果. 将三种优化方法集成在一起后，能够将代码补全的两项指标分别提升 6.76 个百分点和 7.81 个百分点，优化效果最好. 这证明本文提出的优化方法是有效的.

此外，从三种优化方法单独使用以及的排列组合的效果来看，仅使用一种优化方法时，模板提示生成方法在两项评估指标上取得的效果最好；使用两种方法的组合时，模板提示结合二次精排序生成的方式在编辑相似度指标上取得的效果最好，二次精排序结合相似度过滤的方式在完全匹配指标上取得的效果最好，这一

实验结果也印证了本文在经验研究中的启发，即根据输入代码的质量灵活决策是否将检索结果作为提示信息，以及使用提示模板对输入代码和检索结果进行包装，能够提升检索增强代码补全效果.

4.4 代码片段拼接策略影响（RQ3）

本文在 MAGIC 方法的检索库构建阶段设计了代码切分策略，同时构建了对应于每个子代码片段的索引，即通过索引指定每个子代码片段对应的下一个子代码片段的id，从而保证在进行代码切分后的检索库依然保持子代码片段之间的上下文关系，用于在检索精排阶段利用索引来根据检索结果找到对应的下文代码片段拼接起来，补充单个子代码片段缺失的上下文信息. 本实验的目的是对比在检索精排阶段不同的子代码上下文拼接策略对于最终检索增强效果的影响.

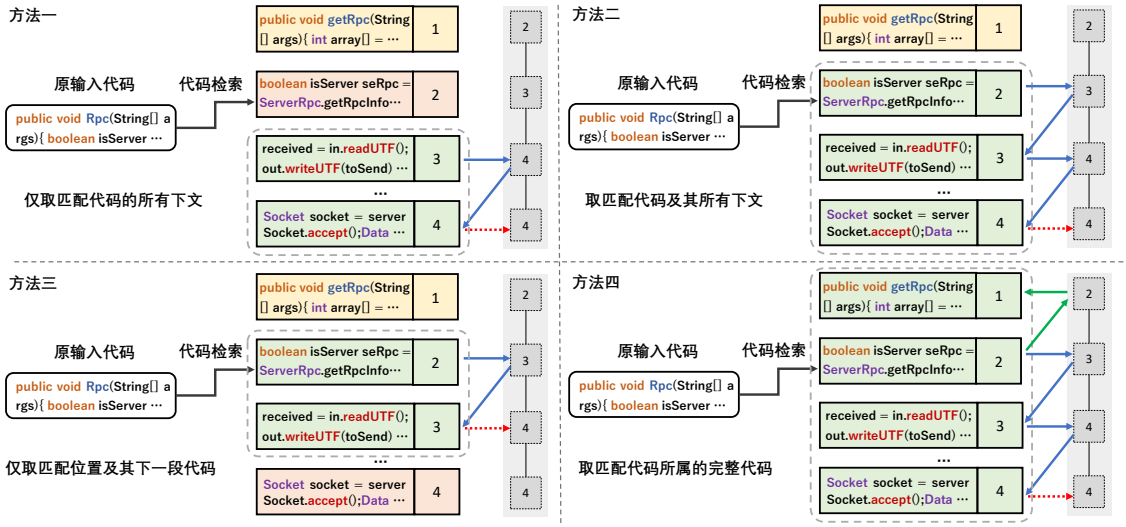

图 11 不同的基于索引的代码拼接策略

本文在设计基于索引的代码拼接策略时，尝试了四种不同的代码上下文拼接策略如图 11 所示，其中的方法二是本文最终采纳的代码上下文拼接策略. 图中从原输入代码出发的黑色箭头指向的位置，指的是从代码检索库中检索到的与原输入代码相似度最高的子代码片段，也即原输入代码经过代码检索得到的第一段匹配代码；灰色虚线框住的浅绿色部分代表经过片段拼接后，作为检索精排阶段的最终输出结果的子代码片段；浅红色部分代表虽然是匹配代码片段对应的代码下文，但未被选中的子代码片段；浅黄色部分代表未被选中的代码上文.

本实验分别实现图 11 中的四种代码上下文拼接策略，对比不同的拼接策略对编辑相似度、完全匹配这两项代码补全质量评估指标的影响. 四种拼接策略的具体实现方法如下：

(1) **方法一：仅取匹配代码片段后面的所有下文**. 这一策略的设计动机是，假设原输入代码通过代码检索匹配到了如图 11 中的id为 2 的子代码片段，那么该子代码片段很可能与原输入代码在内容上是相似的. 代码补全任务是要补全当前输入代码的下文，而匹配代码的下文中可能会包含必要的代码补全位置的下文信息，因此将该id为 2 的匹配位置后面的所有子代码片段重新拼接起来作为检索增强的结果即可，出于节省模型最大输入长度以及降低噪音的考虑，不需要将匹配代码也拼接进来.

(2) **方法二：取匹配代码片段和该片段的所有下文**. 这一策略考虑到检索得到的匹配代码中也有可能包含模型生成所需的下文信息，因此是在方法一的基础上，将检索得到的匹配代码也拼接起来作为检索增强的结果. 即在检索到与原输入代码相似度得分最高的第一个子代码片段后，将该子代码片段及其后面的所有从索引列表中能够检索到的下文子代码片段都拼接起来. 方法二也是本文最终设计采纳的方法.

(3) **方法三：仅取匹配代码片段和它的下一段代码片段**. 这一策略是为了尽可能降低检索增强策略的代码占用模型的最大输入长度而设计的，仅把检索得到的匹配代码和该代码对应的下一个子代码片段重新拼接起来. 例如对于图 11 中id为 2 的子代码片段，则仅将 id 为 3 的代码片段与 id 为 2 的代码片段拼接起来作为检索增强信息.

(4) **方法四：取匹配代码片段对应的原始完整代码**. 这一策略是为了保证输入给模型更完整的提示信息，在检索得到匹配代码片段后，不仅将匹配片段对应代码下文拼接进来，也基于索引反向查找得到该片段对应的代码上文，从而确保模型接收到的检索增强信息是一段具有完整上下文的代码.

本文在不改变 MAGIC 方法的其他配置和超参数的情况下，仅替换代码上下文拼接策略进行实验. 不同的代码上下文拼接策略的评估结果如表 9 所示. 表中方法二即本文设计的取匹配代码和该代码对应的所有下文的拼接方式，方法二在编辑相似度和完全匹配指标上均保持最高的分数，证明了本文设计的代码上下文拼接方法能够有效提升检索增强效果.

从表中可以看出，方法一仅取匹配代码后面的所有下文片段的策略效果最差，这可能是因为匹配位置的代码片段同样包含了代码补全所需要的上下文信息，而不完整的上下文信息导致检索结果不能起到较好的提示作用. 因此子代码片段的补充上下文信息仍然需要从匹配代码的位置开始获取. 此外，取匹配代码对应的原始完整代码的策略(方法四)在编辑相似度和完全匹配两项指标上相比于本文方法仅仅低 0.79 个百分点和 0.66 个百分点，实际上差距不大，这也证明在执行检索增强策略时，补充更丰富的上下文代码语义信息能够在一定程度上提升生成模型的代码质量.

表 9 不同的代码上下文拼接策略评估结果

评估指标	无上下文拼接	方法一	方法二	方法三	方法四
ES	68.01	68.59	**70.27**	68.84	69.72
EM	40.25	40.99	**42.65**	41.12	42.37

4.5 与代码大模型的性能开销对比（RQ4）

本文在 RQ1 中的实验结果表明，本文提出的 MAGIC 方法能够在 124M 的 CodeGPT 模型上达到超过 6B 的 CodeGen 代码大模型的代码补全效果. 为了对比 MAGIC 方法与 CodeGen 代码大模型的性能开销情况，我们计算了在 GitHub Java Corpus 数据集上使用几种不同参数量的 CodeGen 模型和使用 MAGIC 方法所消耗的显存、内存用量以及单条代码片段推理所需的时间，得到结果如表 5 的后三列所示.

从表 5 中可以看出，虽然 CodeGen-16B 大模型的完全匹配指标略高于 MAGIC 方法(高出 1.6 个百分点)，但 MAGIC 方法所需的单条代码补全推理时间仅为 CodeGen-16B 模型推理时间的 17.97%，消耗的显存仅为 CodeGen-16B 模型的 3.50%，消耗的内存仅为 CodeGen-16B 模型的 49.23%. 因此相比于参数量为 CodeGPT 参数量的 483 倍的 CodeGen 代码大模型，MAGIC 方法的优势在于占用的显存和内存更少、单条推理时间更短，在性能和准确率上的表现更加均衡，适合应用于算力有限、不足以支持大模型推理的平台.

5 有效性威胁

本文基于经验研究提出的多阶段改进检索增强的代码补全方法主要解决了检索增强模块可能无法有效引导代码模型生成的问题，以及解决了小参数量模型无法达到与代码大模型接近的推理效果的问题. 分析认为，本文工作可能面临以下两个有效性威胁.

第一个有效性威胁是，本文方法以微软提出的 ReACC 检索增强代码补全框架作为对比方法，但该方法以 faiss 向量检索引擎作为代码检索工具，而本文方法仅仅使用 ElasticSearch 检索引擎作为代码检索工具，不同的检索工具可能存在检索效果上的差异. 为了减轻该威胁，我们在实验部分不但在复现对比方法时直接使用了该工作的论文作者官方提供的代码实现和对应的检索工具，还尝试了使用检索效果远远优于 faiss 的阿里 proxima 向量检索工具，取其中更高的评估结果作为与本文方法的对比.

第二个有效性威胁是，本文的检索增强方法需要基于代码检索库，因此替换更大规模的代码检索库后有

可能取得比本文方法更好的效果. 为了减轻这个威胁，本文从 GitHub 平台共 45,196 个开发者的 145,532 个代码仓库中收集了 968 万条代码，构建出包含 1480 万条代码片段的检索语料库，并在第 4.3 节的实验一中尝试了通过代码切分构建包含 3.5 亿条代码片段的超大规模检索语料库进行对比实验.

第三个有效性威胁是，本文使用了 GitHub Java Corpus 数据集进行模型训练和测试，该数据集和本文构建的大规模代码检索库的数据源都来自 GitHub，因此可能会存在一定的数据泄露威胁，影响本文方法的泛化性. 为了减轻这个威胁，本文在实验前预先对数据集和代码检索库做了去重处理，过滤掉完全相同的代码，并在实际实验中进行了核验，确认未出现检索结果与测试数据重叠的情况；此外，本文在评估实验中使用的 CodeGen 模型的训练数据来自 Google BigQuery，该数据集中可能包含来自 GitHub 的代码，故而本文在构建测试集时预先排除掉了测试集中与 BigQuery 数据集相同的代码.

第四个有效性威胁是，本文所有实验均在阿里巴巴集团实际的生产环境下基于企业级开发工具完成，方法的通用性可能会受到不同场景下的实际运行环境的影响. 为了减轻这个威胁，本文一方面使用公开基准数据集 CodeXGLUE 进行模型的训练评估，并从开源平台的代码库提取项目代码用于构建大规模代码检索库，确保本文方法在数据构建上的通用性；另一方面使用通用的基础软件及版本，其中用于代码检索增强的基础搜索引擎采用 ElasticSearch 8.6 版本，深度学习框架使用 Pytorch 1.9 版本，Python 使用 3.8 版本，均是目前软件开发和深度学习领域主流的开发环境所使用的软件及版本，使本文方法所需环境易于在其他平台上复现. 此外，本文实验使用的阿里云 ECS 服务器是目前国内被工业界广泛使用的服务器，与本文实验相同的服务器配置均可以从阿里云官网购得，因此实验效果更具有实际应用的参考意义.

6 总结与展望

本文通过经验研究，重新审视了代码补全方法中的检索增强策略，重点识别了检索增强方法中影响代码补全效果的三个因素：代码检索库存储的代码粒度、代码检索方法、检索结果的后处理方法. 并针对这三个因素分别用半定性半定量的实验得到可能的优化改进思路. 在此基础上，通过分阶段优化代码检索策略来改进检索增强的代码补全的方法 MAGIC，主要可以分为三个阶段：检索库构建阶段、检索精排阶段、提示生成阶段. 在检索库构建阶段，我们的主要目的是通过代码的预处理和代码切分来提高代码检索库的规模与质量. 首先从开源代码仓库中提取方法级 Java 代码构建初始代码检索库，统计提取出的 Java 方法的长度情况，根据其概率分布确定代码的切分长度. 然后按照固定的步长将方法级代码切分为片段级代码，并利用记录索引的方式保留切分后的代码上下文，扩增代码检索库的规模. 在检索精排阶段，利用排序算法获得与当前输入代码最匹配的代码片段. 首先，基于字符串级别的 BM25 搜索算法，从检索库中检索出前k个候选代码片段. 接着，根据检索的 BM25 分数对候选项进行粗排序，并记录每个选项对应的排名. 最后基于字符串的 Ratcliff 结构相似度和深度学习模型计算的向量相似度的融合分数进行二次精排序，取得与输入代码匹配度最高的检索结果，获取到有效的检索增强信息. 在提示生成阶段，将与输入代码片段之间相似度高的检索结果代码添加到我们预先设计的提示模板中，与输入代码拼接后输入给代码生成模型；将相似度低的检索结果筛去，仅添加空的提示模板与输入代码拼接后，输入给代码生成模型，将有效的代码补全上下文信息传递给生成模型.

实验对比结果表明，本文目前提出的优化方法在评估指标上优于微软提出的 ReACC 检索增强代码补全方法，能够将代码补全的两项指标分别提升 6.76 个百分点和 7.81 个百分点，并能在 124M 参数量的模型上达到超过 CodeGen-2B 与 CodeGen-6B 代码大模型的代码补全效果、且与 CodeGen-16B 代码大模型的代码补全效果接近；同时在代码补全速度和有效节约显存资源方面均能够达到超过 CodeGen 代码大模型的效果.

本文提出的代码补全方法是在大规模的 Java 开源代码上进行研究的，未来可以考虑根据不同编程语言的特性，利用不同的数据处理方法构建出不同的代码检索库. 同时，也可以考虑将更多的项目信息融入代码检索库中，利用条件过滤或特征加权的方式，进一步提升代码检索的效果. 在未来的工作中，我们将研究如何更好地将检索增强方法融入到模型训练阶段，使得模型能够更准确地、更高效地推荐用户所需要的代码.

目前，随着大语言模型技术的发展，已经出现了诸如 1.76T 参数量的 ChatGPT-4 模型[37]和 70B 参数量的

Code Llama 模型[38]等具有巨大的参数规模和先进的生成能力的大模型，然而大模型常面临幻觉问题，生成与输入内容不符或与知识事理不符的内容[39]. Li 等人[40]通过经验研究发现利用检索增强的方式能够缓解 ChatGPT 的幻觉，减少生成内容中的事实错误；Béchard 等人[41]以 110M 参数量的 MPNet-base 作为检索器模型进行检索增强，来辅助 CodeLlama-7B 模型生成 JSON 代码，在提升推理性能的同时有效缓解了大模型的幻觉，该种方式与本文方法类似，但仅使用了基于余弦相似度的向量检索方式，没有使用多种方法进行混合检索，且没有考虑检索库的不同配置对于检索增强效果的影响. 因此未来我们计划探索将本文提出的检索方法以及检索增强相关的结论迁移到多种不同类型的数据库上，例如自然语言的软件开发文档、JSON 对象格式的文档等；并将本文方法应用到更大参数量的大模型上，重点提升大模型生成的代码结果在逻辑严密性和安全性等方面的能力，进一步提升软件开发的效率.

致谢

本工作由阿里巴巴集团通过阿里巴巴研究型实习项目支持.

References:

[1] Yang B, Zhang N, Li SP, Xia X. Survey of intelligent code completion. Ruan Jian Xue Bao/Journal of Software, 2020,31(5):1435−1453 (in Chinese). http://www.jos.org.cn/1000-9825/5966.htm [doi: 10.13328/j.cnki.jos.005966]

[2] Hindle A, Barr ET, Gabel M, Su Z, Devanbu P. On the naturalness of software. Communications of the ACM, 2016, 59(5): 122-131.

[3] Raychev V, Vechev M, Yahav E. Code completion with statistical language models. InProceedings of the 35th ACM SIGPLAN conference on programming language design and implementation, 2014. 419-428.

[4] Liu BB, Dong W, Wang J. Survey on intelligent search and construction methods of program. Ruan Jian Xue Bao/Journal of Software, 2018,29(8):2180−2197 (in Chinese). http://www.jos.org.cn/1000-9825/5529.htm [doi: 10.13328/j.cnki.jos.005529]

[5] Bhoopchand A, Rocktäschel T, Barr E, Riedel S. Learning python code suggestion with a sparse pointer network. arXiv preprint arXiv:1611.08307, 2016.

[6] Chen M, Tworek J, Jun H, Yuan Q, Pinto HP, Kaplan J, Edwards H, Burda Y, Joseph N, Brockman G, Ray A. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374, 2021.

[7] Lewis P, Perez E, Piktus A, Petroni F, Karpukhin V, Goyal N, Küttler H, Lewis M, Yih WT, Rocktäschel T, Riedel S. Retrieval-augmented generation for knowledge-intensive nlp tasks. Advances in Neural Information Processing Systems. 2020,33:9459-74.

[8] Zhang T, Yang D, Lopes C, Kim M. Analyzing and supporting adaptation of online code examples. In2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019. 316-327.

[9] Lu S, Duan N, Han H, Guo D, Hwang SW, Svyatkovskiy A. Reacc: A retrieval-augmented code completion framework. arXiv preprint arXiv:2203.07722, 2022.

[10] Borgeaud, Sebastian, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche et al. Improving language models by retrieving from trillions of tokens. In International conference on machine learning, pp. 2206-2240. PMLR, 2022.

[11] Chen, Tong, Hongwei Wang, Sihao Chen, Wenhao Yu, Kaixin Ma, Xinran Zhao, Dong Yu, and Hongming Zhang. Dense X Retrieval: What Retrieval Granularity Should We Use?. arXiv preprint arXiv:2312.06648, 2023.

[12] Ram, Ori, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, Kevin Leyton-Brown, and Yoav Shoham. In-context retrieval-augmented language models. arXiv preprint arXiv:2302.00083, 2023.

[13] Luan S, Yang D, Barnaby C, Sen K, Chandra S. Aroma: Code recommendation via structural code search. Proceedings of the ACM on Programming Languages. 2019, 3(OOPSLA):1-28.

[14] Zhang F, Chen B, Zhang Y, Liu J, Zan D, Mao Y, Lou JG, Chen W. Repocoder: Repository-level code completion through iterative retrieval and generation. arXiv preprint arXiv:2303.12570, 2023.

[15] Hashimoto TB, Guu K, Oren Y, Liang PS. A retrieve-and-edit framework for predicting structured outputs. Advances in Neural Information Processing Systems. 2018, 31.

[16] Hayati SA, Olivier R, Avvaru P, Yin P, Tomasic A, Neubig G. Retrieval-based neural code generation. arXiv preprint arXiv:1808.10025, 2018.

[17] Xu FF, Jiang Z, Yin P, Vasilescu B, Neubig G. Incorporating external knowledge through pre-training for natural language to code generation. arXiv preprint arXiv:2004.09015, 2020.

[18] Parvez, Md Rizwan, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Retrieval augmented code generation and summarization. arXiv preprint arXiv:2108.11601, 2021.

[19] Liu, Shangqing, Yu Chen, Xiaofei Xie, Jingkai Siow, and Yang Liu. Retrieval-augmented generation for code summarization via hybrid gnn. arXiv preprint arXiv:2006.05405, 2020.

[20] Loukas, Andreas. What graph neural networks cannot learn: depth vs width. arXiv preprint arXiv:1907.03199, 2019.

[21] Li, Jia, Yongmin Li, Ge Li, Xing Hu, Xin Xia, and Zhi Jin. Editsum: A retrieve-and-edit framework for source code summarization. In 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2022. 155-166.

[22] Yu, Chi, Guang Yang, Xiang Chen, Ke Liu, and Yanlin Zhou. Bashexplainer: Retrieval-augmented bash code comment generation based on fine-tuned codebert. In 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2022. 82-93.

[23] Liu, Shangqing, Cuiyun Gao, Sen Chen, Lun Yiu Nie, and Yang Liu. ATOM: Commit message generation based on abstract syntax tree and hybrid ranking. IEEE Transactions on Software Engineering, 2020. 48(5):1800-1817.

[24] Wang, Haoye, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. Context-aware retrieval-based deep commit message generation. ACM Transactions on Software Engineering and Methodology (TOSEM), 2021. 30(4): 1-30.

[25] Shi, Ensheng, Yanlin Wang, Wei Tao, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. RACE: Retrieval-Augmented Commit Message Generation. arXiv preprint arXiv:2203.02700, 2022.

[26] Reimers N, Gurevych I. Making monolingual sentence embeddings multilingual using knowledge distillation. arXiv preprint arXiv:2004.09813, 2020.

[27] Alibaba. Proxima bilin engine. https://github.com/alibaba/proxima, 2022.

[28] Reimers N, Gurevych I. Sentence-bert: Sentence embeddings using siamese bert-networks. arXiv preprint arXiv:1908.10084, 2019.

[29] Peña FJ, Gonzalez AL, Pashami S, Al-Shishtawy A, Payberah AH. Siambert: Siamese Bert-based Code Search. In2022 Swedish Artificial Intelligence Society Workshop (SAIS), 2022. 1-7.

[30] Feng, Zhangyin, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou et al. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155, 2020.

[31] Wang, Yue, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859, 2021.

[32] Song, Kaitao, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mpnet: Masked and permuted pre-training for language understanding. Advances in Neural Information Processing Systems, 2020. 33: 16857-16867.

[33] Lu, Shuai, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv preprint arXiv:2102.04664, 2021.

[34] Reimers. SentenceTransformers Documentation. https://www.sbert.net/docs/pretrained_models.html, 2022.

[35] Radford, Alec, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners, 2019. 1(8): 9.

[36] Nijkamp, Erik, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint arXiv:2203.13474, 2022.

[37] Achiam, Josh, et al. Gpt-4 technical report. arXiv preprint arXiv:2303.08774, 2023.

[38] Roziere B, Gehring J, Gloeckle F, et al. Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950, 2023.

[39] Ji, Ziwei, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. ACM Computing Surveys, 2023. 55(12):1-38.

[40] Li, Junyi, Xiaoxue Cheng, Wayne Xin Zhao, Jian-Yun Nie, and Ji-Rong Wen. Halueval: A large-scale hallucination evaluation benchmark for large language models. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pp. 6449-6464. 2023.

[41] Béchard, Patrice, and Orlando Marquez Ayala. Reducing hallucination in structured outputs via Retrieval-Augmented Generation. arXiv preprint arXiv:2404.08189, 2024.

附中文参考文献:

[1] 杨博,张能,李善平等.智能代码补全研究综述.软件学报,2020,31(05):1435-1453. http://www.jos.org.cn/1000-9825/5966.htm [doi: 10.13328/j.cnki.jos.005966]

[4] 刘斌斌,董威,王戟.智能化的程序搜索与构造方法综述.软件学报,2018,29(8):2180−2197. http://www.jos.org. cn/1000-9825/5529.htm [doi: 10.13328/j.cnki.jos.005529]

邹佰翰(1998－), 男, 硕士生,主要研究领域为代码大数据与智能化软件开发.

汪莹(2000－), 女, 硕士生,主要研究领域为智能化软件开发.

彭鑫(1979－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为智能化软件开发, 云原生与智能化运维, 泛在计算软件系统.

娄一翎(1993－), 女, 博士, 青年副研究员, 硕士生导师,主要研究领域为智能化软件工程、软件测试与分析.

刘力华（1989－），男，阿里云技术专家，主要研究领域为代码智能检查、代码智能编码、代码智能评审等.

张昕东（1994－），男，阿里云技术专家，主要研究领域为软件工程，程序分析，代码生成.

林帆（1988－），男，阿里云高级技术专家，主要研究领域为 DevOps、软件工程相关领域的智能化.

刘名威(1994－), 男, 博士, CCF 专业会员, 主要研究领域为智能化软件开发.