软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software [doi: 10.13328/j.cnki.jos.007228]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于代码克隆差异分析的函数模板挖掘和检索方法*

肖泉彬[1,2], 陈 源[1,2], 吴毅坚[1,2], 彭 鑫[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)

通信作者: 吴毅坚, E-mail: wuyijian@fudan.edu.cn

摘 要: 在软件工程领域, 代码库承载着丰富的知识资源, 可以为开发者提供编程实践的案例参考. 源代码中频繁出现的模式化重复片段, 若能以代码模板的形式有效提炼, 便能显著提升编程效率. 当前实践中, 开发者常常通过源代码搜索复用现有解决方案, 然而此方法往往产生大量相似且冗余的结果, 增加了后续筛选工作的负担. 与此同时, 以克隆代码为基础的模板挖掘技术, 往往未能涵盖由分散小克隆片段构成的广泛模式, 进而限制了模板的实用性. 我们提出了一种基于代码克隆检测的代码模板提取和检索的新方法, 通过拼接多个片段级克隆以及提取和聚合方法级克隆的共享部分, 实现了更高效的函数级代码模板提取, 并解决了模板质量问题. 基于所挖掘的代码模板, 提出了一种代码结构特征的三元组表示法, 有效地对纯文本特征进行补充, 并实现了高效和简洁的结构表示. 此外, 我们也提出了一种结构和文本检索相结合的模板特征检索方法, 以便通过匹配编程上下文的特征来检索这些模板. 基于本方法实现的工具 CodeSculptor, 在包含 45 个高质量 Java 开源项目的代码库测试中展现了其提取高质量代码模板的显著能力. 结果表明该工具挖掘的模板平均可实现 60.87% 的代码量减少, 且有 92.09% 是通过拼接片段级克隆产生的, 这一比例的模板在传统的方法是无法识别出的, 这印证了该方法在识别和构建代码模板方面的卓越性能. 在代码模板检索和推荐的实验中, Top-5 检索结果精确度达到了 96.87%. 我们还通过对随机选择的 9 600 个模板进行的初步案例研究, 讨论了模板的实用性, 并发现大多数抽样代码模板在语义上是完整的, 少数无意义的模板表明我们的模板提取工作的未来潜力. 我们的用户研究进一步表明, 使用 CodeSculptor 能够更有效率完成代码开发任务.

关键词: 克隆检测; 代码检索; 特征表示; 软件开发; 代码复用

中图法分类号: TP311

中文引用格式: 肖泉彬, 陈源, 吴毅坚, 彭鑫. 基于代码克隆差异分析的函数模板挖掘和检索方法. 软件学报. http://www.jos.org.cn/1000-9825/7228.htm

英文引用格式: Xiao QB, Chen Y, Wu YJ, Peng X. Function Template Mining and Retrieval Based on Code Clone Difference Analysis. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7228.htm

Function Template Mining and Retrieval Based on Code Clone Difference Analysis

XIAO Quan-Bin[1,2], CHEN Yuan[1,2], WU Yi-Jian[1,2], PENG Xin[1,2]

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

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

Abstract: In the field of software engineering, code repositories contain a wealth of knowledge resources, which can provide developers with examples of programming practices. If repetitive patterns, frequently occurring in source code, can be effectively extracted in the form of code templates, programming efficiency could be significantly improved. In current practice, developers often reuse existing solutions by searching through source code. However, this method typically generates a large number of similar and redundant results,

* 基金项目: 国家自然科学基金 (62172099)
收稿时间: 2024-01-26; 修改时间: 2024-04-07, 2024-05-10; 采用时间: 2024-05-15; jos 在线出版时间: 2024-07-03

increasing the burden of subsequent filtering. Moreover, template mining techniques based on cloned code often fail to cover extensive patterns constructed from dispersed small clones, thereby limiting the practicality of the templates. A new method is proposed for extracting and retrieving code templates based on code clone detection. This method achieves more efficient function-level code template extraction by stitching together multiple fragment-level clones and extracting and aggregating the shared parts of method-level clones and addresses the issue of template quality. Based on the mined code templates, this study comes up with a triplet representation method for code structural features that effectively supplements plain text features, and implements an efficient and concise structural representation. In addition, this study presents a template feature retrieval method that combines structural and textual search to retrieve these templates by matching features of the programming context. The tool implemented based on this method, CodeSculptor, demonstrates its significant capability to extract high-quality code templates in a test against a codebase containing 45 high-quality Java open-source projects. The results show that the templates mined by the tool achieve an average code reduction of 60.87%, with 92.09% produced by stitching fragment-level clones, a proportion of templates that is not identifiable by traditional method., It proves the superior performance of the method in recognizing and constructing code templates. Furthermore, the accuracy of the top-5 search results in our code template search and recommendation is 96.87%. A preliminary case study on 9600 randomly selected templates reveals that most of the sampled code templates are complete and coherent in semantics, thus affirming their practicality. Nonetheless, there are a few meaningless templates, highlighting the future potential to refine the proposed template extraction strategy. The user research further shows that code development tasks can be done more efficiently with CodeSculptor.

Key words: clone detection; code search; feature representation; software development; code reuse

克隆代码, 指的是在软件系统中相同或近似的代码片段. 研究表明, 在开源和企业的代码库中, 克隆代码所占比例在 20%-80% 之间[1][2]. 这些克隆代码片段, 往往实现了相同或相近的功能, 体现了类似功能的模式化实现, 隐含了软件开发的共性知识. 现有的研究能对这些模式化的代码实现进行分析挖掘, 构建代码模板[3], 从而支撑代码检索、代码推荐等软件开发需求.

代码检索、代码补全和重构推荐等过程都在一定程度上依赖于对代码模板的复用. 代码搜索主要是在编程语言的语义和句法空间内, 对查询的代码片段进行匹配, 以寻找已存在的相关代码段或模块. 代码补全则更侧重于实时性, 在开发过程中, 根据已输入的代码片段, 它会自动推荐和输入相对应的剩余部分, 就像是在输入过程中动态进行的代码搜索, 其背后也是代码模板的运用. 至于重构推荐, 它的目标是在保持功能不变的情况下, 改善代码的内部结构. 为达成这一目标, 它需要识别出代码中的糟糕模式, 并在代码模板中推荐一个更清晰、更易维护的改进方案.

现有的代码检索工具[4], 如 FaCoY[5]、AROMA[1]和 Sentas[6], 虽然它们通过允许开发者使用部分代码片段作为输入在代码库中检索相关代码, 但存在的问题是 FaCoY 返回大量的相关代码片段而未消除相似片段, AROMA 和 Sentas 则可能返回冗长且难以概括的代码, 这将导致开发者在理解和再利用相关代码片段的难度增加. 另一类工具是基于模式的代码补全工具[7][8], 通过挖掘大型代码语料库中的常见 API 使用模式, 并依赖这些模式对部分编写的程序提供代码补全建议, 只要部分程序匹配到模式的前缀. 然而, 这些模式挖掘工具受到模式数量的限制, 覆盖相对较小的常见模式子集. 能推荐的代码范围仅限于已挖掘出的模式, 如果开发者的需求超出了已挖掘的模式, 这些工具就无法给出有用的建议. 至于代码克隆检测工具, 其对待查询片段的处理过于死板, 通常检索与查询片段完全一致的代码片段, 缺乏灵活性, 无法为扩展查询片段而提供进一步的信息.

在本研究中, 我们对代码模板的定义不仅包括了具备共性知识的克隆代码片段, 而且强调了这些模板的可配置性和在多种开发上下文中的复用性. 我们特别关注那些在功能上界限明确、能反映常见编程任务和问题解决模式的可重用代码结构. 在此基础上, 我们提出一种新颖的代码模板提取和推荐方法, 并设计了命名为 CodeSculptor 的工具进行实现. 该方法通过提取和聚合方法级克隆的共享部分, 有效地解决了模板冗余问题. 首先使用代码克隆检测器在库间检测代码克隆, 然后从具有功能差异的方法中提取公共上下文, 从而产生代码模板. 在克隆检测环节, 我们进行了片段级和方法级的双重检测. 对于方法级别的克隆, 我们通过差异性分析并引入投票机制进行模板生成. 而对于片段级别的克隆, 则被汇总为大间距的代码克隆[9], 以便提取出更多实用的开发模式. 此过程提供了一种更高效的方法级代码模板提取手段, 消减了开发者在面对大量重复代码片段时的工作负担. 模板挖掘之后为了实现推荐, 我们考虑了代码的文本和结构两个层面的特征表示. 针对文本特征, 我们着眼于特异性并使用相关算

法[10]进行计算. 我们的结构特征则借鉴并简化了 AROMA 中的近似 AST 方法, 因为关键词已在文本特征中得到体现, 我们只需专注于代码的结构表示, 无需为维护复杂的树形数据结构付出过多计算和存储代价, 更为简洁高效. 我们使用这些特征构建倒排索引, 并用于检索. 其中 BM25 权重值高的特征揭示了包含该关键字的模板与代码片段的相关性更高. 最终, 我们不仅根据匹配特征的数量, 更将文本关键词作为一种重要的特征, 结合其 BM25 权重, 对返回的结果进行精炼排序, 以确保与当前代码片段匹配程度最高的模板优先展示.

为验证模板的实用性、检索的准确性以及在实际应用中的效果, 我们从语言一致性和编码风格一致性的角度出发挑选了 45 个高质量 Java 开源项目作为实验数据集展开测试. 这些测试包括模板聚合程度以及数量统计分析、消融实验、前 k 次检索召回实验以及 20 个具体案例的研究. 结果显示, 我们的方法在这 45 个开源项目中挖掘出了 392, 580 个模板, 这些模板覆盖了 60.87% 的原始代码, 这在一定程度上显示了模板在表示原始代码中的普遍模式和功能方面的有效性. 更关键的是, 通过片段级克隆合并形成的模板占模板总量的 92.09%, 这一部分模板是传统的方法无法识别出的, 这一点明确地体现了我们在提取代码模板方面的优越性. 我们还对 2, 115 个方法代码进行了针对特征匹配的消融实验, 结果表明, 在结构丰富的代码中, 结构特性可以有效地补充文本特性. 对于连续和非连续代码片段的模板推荐, 实现了高检索召回率, 展示了基于特征的检索的效果. 我们还随机选取了 20 个模板进行案例研究, 结果显示 CodeSculptor 在实际开发过程中的积极影响.

本文的其余部分组织如下. 第 1 节介绍背景知识和动机. 第 2 节描述 CodeSculptor 创建模板和提供推荐的方法. 第 3 节介绍了我们实验的设置, 包括研究问题、数据集选择和评估指标, 并根据实验结果回答所有的研究问题. 第 4 节介绍了基于 CodeSculptor 开展的用户研究. 第 5 节给出总结和未来研究展望.

1 背景知识与动机

在软件开发中存在大量的代码重复现象, 有时出于功能的类似, 有时则是因为编程习惯和样式的一致. 然而, 这种重复的代码往往让开发者在修改和维护代码时面临挑战. 此外, 相似功能的代码之间存在的微小差异也增加了理解和复用代码的难度. 例如, 如果我们考虑在 C 语言编程任务中的 fopen 和 fclose 对, 虽然这两者在许多函数中常常一起出现, 但 fopen 和 fclose 之间的其他代码则可能因功能需求的不同而有所差异.

代码示例 1 和 2 展示了两个来自 JMeter①(一个知名的开源 Java 项目) 的方法代码片段. 这两个方法都在主框架中显示一个消息对话框以响应某些特定动作事件. 虽然这两个方法的很多部分是相似的, 如创建对话框、设置其位置、将其添加到主框架, 但是它们在实现特定功能时的代码仍有较大差异.

代码示例 1: JMeter showInputDialog 方法

```
1 private JSyntaxTextArea cURLCommandTA;
2 ...
3 private void showInputDialog(ActionEvent event) {
4     EscapeDialog messageDialog = new EscapeDialog(getParentFrame(event),
5     JMeterUtils.getResString("curl_import"), false);
6     Container contentPane = messageDialog.getContentPane();
7     contentPane.setLayout(new BorderLayout());
8     statusText = new JLabel("", JLabel.CENTER);
9     statusText.setForeground(UIManager.getColor(JMeterUIDefaults.LABEL_ERROR_FOREGROUND));
10    contentPane.add(statusText, BorderLayout.NORTH);
11    cURLCommandTA = JSyntaxTextArea.getInstance(20, 80, false);
```

① https://github.com/apache/jmeter

```
12    cURLCommandTA.setCaretPosition(0);
13    contentPane.add(JTextScrollPane.getInstance(cURLCommandTA), BorderLayout.CENTER);
14    // 11 special lines of code
15    messageDialog.pack();
16    ComponentUtil.centerComponentInComponent(
17        GuiPackage.getInstance().getMainFrame(), messageDialog);
18    SwingUtilities.invokeLater(() -> messageDialog.setVisible(true));
19  }
```

代码示例 2: JMeter showResult 方法

```
1 private static void showResult(ActionEvent event, String result) {
2     EscapeDialog messageDialog = new EscapeDialog(getParentFrame(event),
3           JMeterUtils.getResString("export_transactions_title"), false);
4     Container contentPane = messageDialog.getContentPane();
5     contentPane.setLayout(new BorderLayout());
6     contentPane.add(new JLabel(
7           JMeterUtils.getResString("export_transactions_exported_property"),
8             SwingConstants.CENTER), BorderLayout.NORTH);
9     JSyntaxTextArea syntaxTextArea = JSyntaxTextArea.getInstance(10, 80, true);
10    syntaxTextArea.setCaretPosition(0);
11    contentPane.add(JTextScrollPane.getInstance(syntaxTextArea), BorderLayout.CENTER);
12    messageDialog.pack();
13    ComponentUtil.centerComponentInComponent(
14          GuiPackage.getInstance().getMainFrame(), messageDialog);
15    SwingUtilities.invokeLater(() -> messageDialog.setVisible(true));
16  }
```

正如代码示例 1 中的 showInputDialog 方法所省略的 11 行代码, 这些代码实现了一系列特定的功能. 诸多实际开发场景和上述代码样例相似, 类似功能的代码实现往往会在不同的地方重复出现. 虽然这些重复代码结构相似, 功能相似, 但由于各自处于不同的上下文环境中, 它们在实现特定功能时可能具有微小的变化. 以代码示例 1 为例, 开发者在开发过程中会期望在输入代码

```
private void showInputDialog(ActionEvent event) {
EscapeDialog messageDialog = new EscapeDialog(getParentFrame(event))
```

时, 可以直接返回代码示例 1 中的所有阴影部分代码, 从而减少开发者的键盘输入.

当前, 如何找到之前写过的相似代码以减少重复劳动的问题, 往往需要依赖各种代码检索工具. 然而, 这些工具存在不少问题. 一方面, 它们可能会返回大量的检索结果, 不论检索结果中的内容是否通用, 返回的代码片段由于检索特征包含的缘故又容易过长且难以理解, 增加了使用者的负担. 另一方面, 现有的代码补全工具往往只能对几种预设的模式提供支持, 对于超出这些模式的复杂场景则束手无策, 而上述代码样例中的重复由于其业务特点很难被形容为某一种模式.

在上述案例中, 我们观察到其中存在有 Type-1、Type-2 和 Type-3 这三种类型的代码克隆.

Type-1 克隆: 所有代码文本在片段级别克隆类中都完全相同. 例如, showInputDialog 方法中第 15 行到第 18

行的代码与 *showResult* 方法中第 12 行到第 15 行的代码完全相同. 这是在从它们中提取公共上下文时最容易处理的情况.

Type-2 克隆: 在标记化预处理过程后, 所有代码片段都完全相同. 例如, *showInputDialog* 方法中第 4 行到第 7 行的代码, 与 *showResult* 方法中第 2 行到第 5 行的代码相比, 只有一个不同的字符串. 两个方法中的函数 *getResString* 的参数分别是*"curl_import"* 和 *"export_transactions_title"* , 对应于特定的动作事件类型. 这两个字符串都可以被视为一个字符串字面量标记. 此外, *showInputDialog* 方法中第 13 行的变量 *cURLCommandTA* 实际上与 showResult 方法中第 11 行的变量 *syntaxTextArea* 的类型相同. 它们都可以被视为一个单一的标识符标记.

Type-3 克隆: 在标记化预处理过程后, 所有代码片段都相似. 例如, 在 showInputDialog 方法中, 变量 *cURLCommandTA* 的声明和定义是分开的. 而在 showResult 方法中, 对应变量 *syntaxTextArea* 的声明和定义在一行中, 使 *showResult* 方法中的第 9 行比 *showInputDialog* 方法中的第 11 行多包含一个 token.

那么通过成熟的克隆检测工具, 我们能够识别出这些存在于代码库中的重复出现的代码. 但是单纯使用克隆检测的处理方式, 只能检索出与查询片段一致或相似的代码片段, 在代码样例中, 存在三对不同的片段级克隆代码, 显然如果按片段级克隆为粒度, 在案例中无法帮助开发人员更好的实现代码复用. 因此我们希望能够通过使用某种方法基于克隆检测结果进一步处理并形成代码模板, 这样, 当开发者面临重复编程任务时, 他们只需输入部分代码片段, 便能引用相似的代码模板, 有效提升了编程效率. 尤其是在遍历大型代码库且无法找到合适参考源时, 该工具可以成为开发者的得力助手.

与此同时, 我们也对模板检索的方法进行了一定的探索. 在克隆检测提取出的模板上, 我们考虑了代码的文本和结构特征, 基于此构建倒排索引进行检索. 我们注意到, 重复的代码模式并不总是完全相同, 开发者在查询时, 也需要在可复用的代码片段之间找出精准匹配和模糊匹配的部分. 因此, 我们设计了考虑匹配特征数和 BM25 权重进行排序的策略, 以确保与当前代码片段匹配程度最高的模板优先展示.

综上, 我们的目标是通过有效地使用克隆检测和模板提取技术, 以更灵活、更精确地应对代码重复问题, 进而提升开发者的生产力和代码质量. 我们致力于通过这种方法, 改变现有工具对代码模板提取和检索的处理方式, 将其从僵硬、低效的模式中解放出来, 转变为更灵活、更智能的方式.

然而, 随着处理的克隆实例数量增加, 通过纯程序的方式来提取出完美的公共上下文变得异常困难. 比如应该如何确定从哪些方法中抽取公共上下文, 以我们的经验来看, 如果参与公共上下文提取的函数数量过大, 可能导致生成的模板过于碎片化, 且可能导致关键信息的遗漏. 相反, 如果函数数量过小, 生成的模板可能则包含过多特定业务案例的代码.

而且, 代码模板在代码补全和代码重构过程中有着广泛的应用价值, 我们希望无论开发者输入的是部分代码片段还是完整方法, 他们都能够从中找到符合当前开发任务的模板并进行复用以减少工作量. 然而, 当输入代码片段仅为部分时, 我们可能需要引入更多的文本权重信息来确定关键词的优先级, 当输入的代码片段存在一定的结构时, 则需要代码结构特征来支持更精确的模板检索, 这无疑增加了实现的复杂性, 具体的特征检索效果如何还需要进一步的实际验证和测试.

2 基于克隆差异分析的函数模版挖掘与推荐方法

我们提出 CodeSculptor, 一种从跨项目代码克隆中提取代码模板用于代码检索推荐的新方法. 图 1 概述了该方法的整体框架结构, 涉及从克隆检测到函数模板挖掘, 再到模板推荐的完整流程. 首先, 在函数模板挖掘阶段, 以全面的代码克隆检测为起点, 通过选择合适的克隆检测工具、设定参数以及详尽地分析处理片段级克隆代码, 奠定了挖掘函数模板的基础. 随后详细阐述了函数模板挖掘的过程, 强调在此阶段对检测到的克隆实例进行差异性分析并挖掘出高质量的函数模板. 在构建了模板库后, 我们进一步抽取代码中的关键特征, 并构建特征索引, 为最终的检索流程奠定基础. 如此, 开发者在提出代码片段请求时, CodeSculptor 能迅速响应并检索出高相关性的代码模板, 以有效提升开发效率.

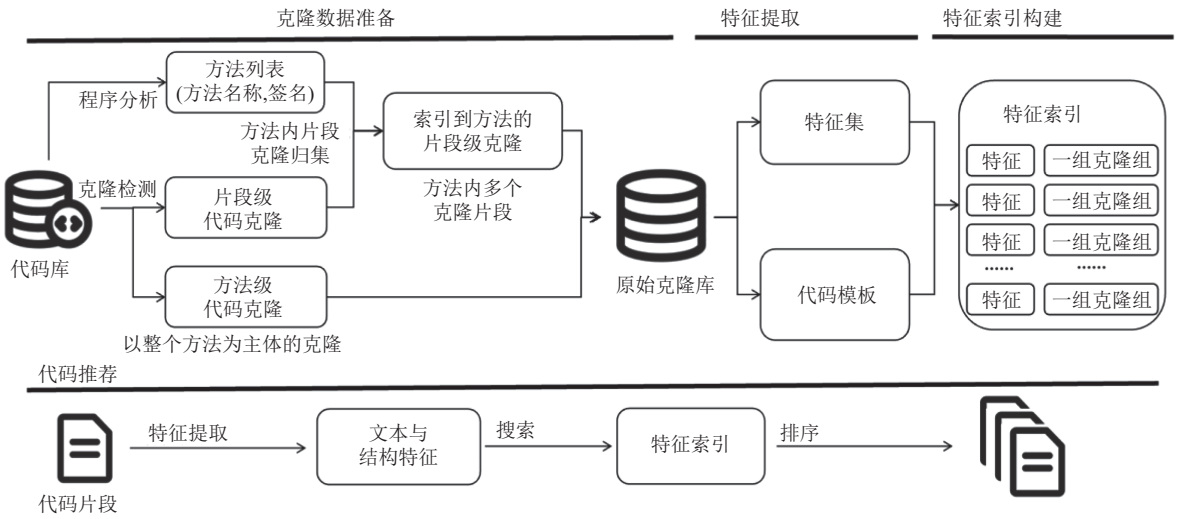

图 1 CodeSculptor 架构图

2.1 克隆检测策略

为了从多个代码仓库中提取公共上下文, 我们首先在所有仓库中检测代码克隆, 然后根据它们的上下文相似性将代码克隆片段进行分组.

2.1.1 克隆检测工具与设置

函数重构推荐的基本粒度为方法级别, 但是方法的共同上下文可能相距较远, 比如资源的申请和释放, 同步加锁等, 而这些上下文本身可以从一定程度上代表一个函数, 即该函数的重要特征, 所以我们在方法级别和片段级别都进行了克隆检测.

在克隆检测工具的选择上, 我们选择了 SAGA[11], 该工具在检测方法级别和片段级别的代码克隆方面表现出非常高的效率, 可以在 11 分钟内检测出 1 亿行代码中的代码克隆. 其基本的工作原理是首先将代码进行符号化处理[12], 然后利用可并行加速的算法构建后缀数组, 该数组揭示出代码中的重复段落. 通过识别这些重复片段, 可以恢复出其所在方法的详细信息, 并据此计算两个方法的相似度.

传统的函数级代码克隆检测任务通常关注多个函数在整体上的文本或结构相似性. 然而, 在提取模板的过程中, 更加关注上下文之间的关联性. 例如 C 语言的文件读取, 需要先调用 *open*() 方法, 最后 *close*() 释放资源. 亦或者是在操作指针前, 需要先进行判空等操作. 但是这些上下文之间可能存在许多差别很大的部分, 它们的共同上下文可能相距较远. 因此, 在单纯函数级克隆检测的基础上, 本文方法还对原始的代码库进行了片段级的克隆检测. 在克隆检测过程中, 特地采用以下设置:

1. 方法级别代码克隆检测的相似度阈值设置为 0.7, 这是一个在召回率和精度之间达到平衡的经验设置. 超过阈值的片段被判定为克隆, 未达到阈值的不被视为克隆.

2. 方法应至少包含 50 个标记, 片段应至少包含 20 个标记. 以往的研究证明, 较小的方法或片段大小可能会导致更高的假阳性.

2.1.2 Type-3 克隆检测处理

Type-3 克隆被视为软件克隆检测中的一种挑战, 主要因为它涉及到代码片段的结构相似性而非严格的文本相似性. 这种克隆类型可能包含被修改的、添加的或删除的语句, 使得在检测和管理时需要更精细的分析工具来识别潜在的同源代码片段.

为了应对这一挑战, 我们引入最小编辑距离 (MED) 来计算标记化后两个标记序列之间的相似性. 编辑距离衡量的是将一个字符串变换成另一个字符串所需进行的最小数量的编辑操作 (涵盖插入、删除或改动字符), 并用动

态规划算法计算以确保效率. 两个标记序列之间的相似性计算公式为:

$$similarity(s_1 s_2) = 1 - \frac{MED(s_1, s_2)}{max(|s_1|, |s_2|)} \tag{1}$$

在这里, $MED(s_1, s_2)$ 表示将序列 s_1 转化成序列 s_2 所需的最小编辑次数, 而 $|s_1|$ 和 $|s_2|$ 分别表示序列的长度, $max(|s_1|, |s_2|)$ 则指的是两个序列长度的最大值. 这个比率衡量了两个序列之间的差异度, 其值越接近 1, 表明两者结构上相似度越高.

当计算得到的相似度超过设定的阈值 0.7 时, 相应的代码片段会被分入同一方法级别的克隆类中, 为后续处理和模板生成提供基础. 这个阈值通常是基于先前的研究、实验观察或者经验选取的[11], 目的是为了区分结构上相关的代码片段与不相关的代码片段. 经过以上策略, Type-3 克隆检测的问题得到了有效的管理, 将复杂的克隆检测过程转换为了一种清晰、可行的解决方案. 同时, 这一方法不仅简化了从大型代码库中识别和提取共享上下文代码的过程, 而且提供了一个坚实的基础, 以促进高效的后续函数模板挖掘.

2.2 函数模板生成

2.2.1 片段级克隆代码的聚合策略

经过克隆检测程序后, 相邻上下文中相似的方法被组织到同一方法级别的克隆类中. 在对片段级克隆代码的分析中, 经常可以发现, 同一方法内部包含的多对克隆代码片段相互隔开, 形成所谓的克隆间隙. 这种现象表现为在方法级别上, 多个相似或相同的代码块被非克隆代码分隔开.

对于在共同上下文中相隔较远的克隆片段所属的方法, 这些方法会继续保留在它们各自的片段级克隆类别中. 进一步地, 包含多个这样片段间隙的方法将被聚集, 从而构成具有更大空间间隔的方法级克隆类别. 这类间隙克隆可以被看作 Type-3 克隆的一种衍生形式, 其中非关联代码的介入导致克隆片段间断, 影响了克隆代码的连续性.

间隙克隆的有效构建策略基于克隆检测的输出结果, 对片段级克隆进行方法级别的整合. 在该过程中, 所有已识别的克隆片段根据它们所在的方法进行分组. 接着, 利用参数化或者占位符的手段连接同一方法内的克隆片段间隙, 从而形成一个含有完整方法上下文的函数模板. 整合的目标是在保持代码原有的逻辑结构和语义的同时, 提供适应性, 以适配不同的应用背景. 多个克隆片段所在的方法集合则被视为大间隙克隆对, 相应地, 大间隙克隆组为处理克隆代码的复合形态提供了框架. 算法 1 概述了上述聚合过程的实现.

算法 1: 片段级克隆聚合算法

输入: *识别的克隆片段集合 $P=\{p_1, p_2, p_m\}$, 克隆检测的阈值 threshold*
输出: *方法级别克隆类集合 C*

```
1 begin
2 C ← ∅
// 根据所属方法对克隆片段进行分组
3 M ← groupByMethod(P)
4 foreach 方法中的片段分组 g_m ∈ M do
5    Template_m ← {}
6    PreviousEnd ← NULL
7    foreach 克隆片段 p ∈ g_m do
8       if PreviousEnd ≠ NULL then
9          Gap ← findGap(PreviousEnd, p)
10         Placeholder ← createPlaceHolder(Gap)
11         Template_m ← Template_m ∪ Placeholder
```

12 $Template_m \leftarrow Template_m \cup p$
13 $PreviousEnd \leftarrow end(p)$
14 $C \leftarrow C \cup createCloneClass(Template_m, threshold)$
15 **return** C

此算法简化了从克隆片段集合到方法级别克隆类的转换过程, 并体现出了如何通过填补间隙来构合完整的克隆函数模板的策略. 它将克隆间隙视为连接点, 通过动态创建占位符, 增加了代码的适应性, 同时保持核心逻辑的一致性. 这是实施大规模克隆管理和重构的有效方法.

2.2.2 克隆差异分析

在基于克隆的模板生成过程中, 处理多克隆实例间的差异, 并考虑间隙克隆的方法对生成高质量的模板至关重要.

首先通过克隆检测技术识别代码中的重复模式, 并将这些克隆归类, 以此提取它们共有的上下文结构, 即函数模板. 为了计算在多个方法体中的公共行及其出现次数, 我们引入了最长公共子序列 (Longest Common Subsequence, LCS) 算法[13]. LCS 算法是一种动态规划算法, 这一算法可以帮助识别模板的公共骨架, 为后续的模板生成打下基础. 为了克服传统 LCS 算法提取多个代码实例间模板时面临的挑战, 诸如若干实例间的微小差异导致提取的公共部分不足, 影响模板的完整性, 本研究创新性地在 LCS 框架下引入了一种打分机制. 此机制的设计旨在更精细地判定代码片段的公共性质.

具体地, 当某代码片段在一组方法中多次出现, 但并非无一例外地存在于所有方法中, 此时便启动打分程序. 考虑的维度包括代码文本的相似度, 即通过最小编辑距离相对于最大片段长度计算而得, 以及代码符号的一致性, 后者关注标识符的通用语义而不拘泥于其实际命名. 例如, 即便两个变量名称各异, 它们在逻辑上扮演相同的角色时, 也应被识别为相关性高. 片段的最终评分受其在方法集合中出现频率的影响, 频繁出现者归于公共部分, 否则归为非公共部分. 这一打分的机制显著提升了模板提取的准确性, 有效地排除了细微差异所带来的干扰, 确保了抽取模板的质量和应用的广泛性.

算法 2: 函数模板生成算法

输入: *方法体 $M=\{m_1, m_2, m_n\}$, 方法体大小 n, 出现阈值 threshold*
输出: *模板行列表 List<Templatelines> $T=[l_1, l_2, ...]$*

1 **begin**
2 *templateLines* ← ∅
3 *occurs* ← ∅
4 **forall** *line l* of m_1 **do**
5 *templateLines.add*(*l*)
6 *occurs.add*(1)
7 **for** $i \leftarrow 2$ to n **do**
8 *lcs* ← *computeLcs*(*templateLines*, *occurs*, m_i)
9 *indices* ← *determineCommonLineIndices*(*lcs*)
10 **forall** *indice* of *indices* **do**
11 *occurs*[*indice*]++
12 (*places*, *indices*) ← *determineUniqueLineIndices*(*lcs*)
13 *methodLines* ← lines of m_i
14 **forall** (*place*, *indice*) of (*places*, *indices*) **do**
15 *templateLines.add*(*place*, *methodLines*[*indice*])

```
16          occurs.add(place, 1)
17 k ← size of templateLines
18 results ← ∅
19 for i ← 1 to k do
20      if(occurs[i]/n) >= threshold then
21          results.add(templateLines[i])
22 return results
```

LCS 算法作为 *computeLCS*(), 模板生成步骤的细节在算法 2 中呈现. 参数 **M** 由每个检测到的克隆类的 **n** 个方法体组成. 参数 *threshold* 是一个超参数, 我们设定阈值 (*threshold*) 为 0.5, 我们假设一个出现频率超过 *threshold,* 即至少存在半数及以上方法体存在的行会被纳入我们生成的模板中, 从而确保我们的模板捕捉到最具代表性和普遍性的代码模式. 在算法 2 的第 1 至第 16 行中我们将 m1 中的每一行添加到 *templateLines* 中, 并将每行的出现次数设为 1. 然后, 我们计算 *templateLines* 和下一个方法体 m_2 之间的 LCS 为 *lcs*. 对于 *templateLines* 中的每一行, 我们都在 *occurs* 中增加它的出现次数. 对于 m_2 中的每一行, 我们将其插入 *templateLines*, 然后在 *occurs* 中增加其出现次数. 在逐个计算 *templateLines* 和剩余方法体之间的 LCS 之后, 我们得到所有方法体的所有行和其出现次数的集合 *templateLines*. 最后在第 17 至第 20 行, 我们筛选出出现频率高于 *threshold* 的行, 得到提取的克隆模板.

值得一提的是, 尽管我们的这种投票机制和 MICoDe 工具[14]有一定的类似之处, 但还是有显着区别. 具体来说, 我们的工作更加侧重于通过 LCS 和打分机制识别和提取出代码模板, 而 MICoDe 的主要目标是通过反向工程和聚类技术找出对应的程序元素, 识别和比较多个代码实例之间的差异, 然后比较和提取出重复出现的设计模式.

2.3 代码特征提取

代码特征是检索和推荐代码模板的重要依据. 所有的代码都按照方法的基本粒度进行分割. 需要识别其通用特性, 如标识符文本和结构, 并为索引做准备.

2.3.1 文本特征提取

首先, 我们对代码中的标识符文本进行特征提取, 旨在捕获具有语义显著性的信息, 例如独特的变量和 API. 在这个过程中, 我们主要处理非关键字文本内容, 包括方法名、变量名和函数名, 以获取有关方法初始语义信息的线索.

具体而言, 我们将每个方法视为一个文档, 其中的非关键字文本内容被视为词. 我们考虑词的长度大于 *'n'*(例如, 取 *'n'* 为 3, 以排除过短的词可能对分析造成的干扰), 并在对标识符进行驼峰和下划线分割后将符合条件的词包含在词列表中. 接下来, 我们计算这些词的 BM25 值. TF-IDF 和 BM25 都是用于文本信息检索和排名的算法. BM25 引入了一个额外的饱和函数, 以处理长文档和短文档之间的差异. 较高的 BM25 权重表示包含该关键字的模板与代码片段的相关性更高, 即该关键字更可能代表与检索相关的代码片段. 通过这种方法, 我们能够更有效地表达方法的特性, 尤其是那些独特的词. 选择性良好的特征词将优先包含在方法的特征索引中, 从而提高对方法的语义描述的准确性.

2.3.2 结构特征提取

从代码中提取结构信息的过程中, 我们采用了程序分析技术 Tree-sitter①, 这是一种应用了"Croncrete Syntax Tree"(CST) 技术的解析库. 与抽象语法 (Abstract Syntax Tree, AST) 相比, CST 在处理实际代码中显得更为灵活高效, 它能够形成一个只需依赖代码的具体语法结构即可生成的树形结构. 因此, 即便代码无法编译成功, 我们都能对其进行处理, 从代码中解析出结构信息.

我们的研究中对代码结构特征的提取, 参考并简化了 AROMA 中的近似 AST 方法. AROMA 中的结构特征表

① https://tree-sitter.github.io/tree-sitter/

示方法被称为"Simplified Parse Tree Representation". 在这种方法中, 每个内部节点代表一个简化的解析树, 并使用自定义的树标签进行标记. 如果一个元素是关键字标签, 它将与标签的值连接在一起. 如果一个元素是非关键字标签或简化的解析树, 它将与特殊符号 "#" 连接. 与 AROMA 的方法相比, 我们进行了更进一步的精简. 我们特别关注的语法结构包括 for 循环、while 循环、if 判断、try-catch 等结构, 以及嵌套循环等. 通过遍历代码结构信息, 设计了三元组的的节点特征类型, 包括父-子节点对或祖先-后代节点对.

代码示例 3: 结构特征表示

```
public void structureDemo() {
        for (int i = 0; i < 10; i++) { // (for1, 1, ⊥)
            if (i % 2 == 0) { // (if1, 1, for1)
                for (int j = 0; j < 5; j++) { // (for2, 1, if1) (for2, 2, for1)
                        // do something
                }
            }
        }
}
```

具体而言, 代码中的特定文本或语法关键字在这个表示中以三元组 (当前符号, 向上层次, 目标符号) 的形式存在. 例如: **(for1, 1, ⊥)(if1, 1, for1)(for2, 2, for1)** 表示一个方法中有一个 for 循环, 里面带有一个 if 判断, if 判断中还有一个 for 循环, 具有这种结构的代码如代码示例 3 所示.

2.3.3 代码特征对比

在上述内容中, 我们深入探讨了代码特征表达方式, 尤其是我们提出的三元组表示法与 AROMA 的特征表示方法之间的差异. 为了清晰呈现这一对比, 参照代码示例 4 的代码片段, 可视化结构特表示方法如图 2.

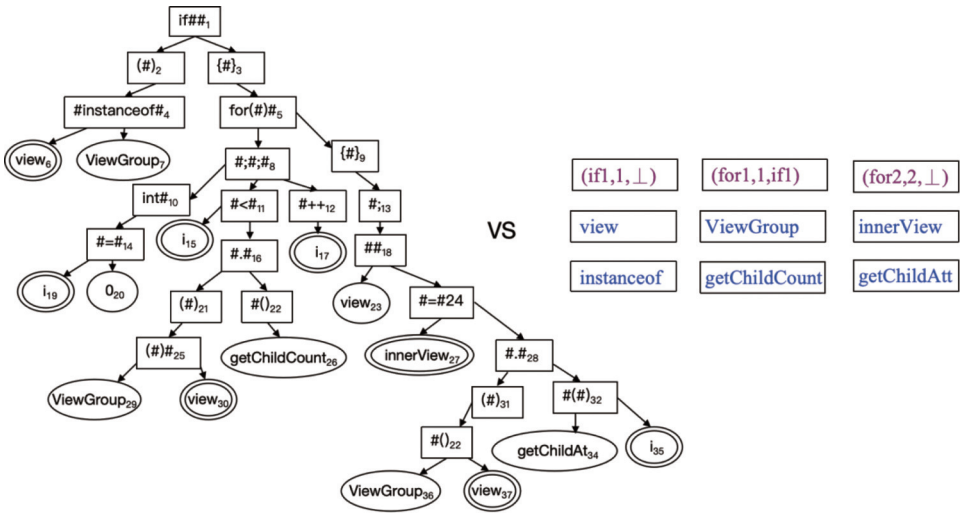

图 2 特征表示方法对比

代码示例 4: 特征表示方法样例

```
if (view instanceof ViewGroup) { // (if1, 1, ⊥)
```

```
for (int i=0; i < ((ViewGroup) view).getChildCount(); i++) { // (for1, 1, if1) (for1, 2, ⊥)
    View innerView = ((ViewGroup) view).getChildAtt(i);
    }
}
```

AROMA 的特征表示依赖于抽象语法树 (AST) 的节点以及它们之间的关系, 这种方法在捕捉代码结构上非常详尽, 但它要求输入文件必须能够正确编译, 并且需要对抽象语法树进行完整的遍历. 相对而言, 我们的三元组表示法提供了一种更加高效和简洁的结构特征表示方法. 它不仅消除了需要 AST 的编译限制, 而且避免了保持复杂树形结构的需求, 从而从根本上减轻了计算和存储方面的压力.

我们对代码结构的捕捉不在于传统的关键字匹配, 而是将关注点放在了代码的层次关系和流控制结构上, 这是因为这些信息已经充分体现在了文本特征之中. 三元组采用形如 (当前符号、上下层次、目标符号) 的格式, 将复杂的代码结构信息压缩至简单的列表表示, 这不仅有助于更快地遍历和处理, 也为基于结构的代码推荐提供了必要的灵活性和精确度.

此外, 该三元组表示法还极大地优化了检索索引的构建过程, 由于索引仅需涵盖表示结构关系的三元组列表而非完整的树形结构, 这使得索引更加紧凑, 提升了检索效率.

2.4 特征索引构建

在像上述那样对提取的代码特征信息进行标准化处理后, 我们需要为与代码克隆相关联的每个特征建立索引. 对于文本和结构特征, 我们可以通过创建"特征-克隆组"映射表来实现. 这样一来, 我们能够迅速定位相应克隆组的特征, 便于后续的分析和查询.

基于文本特征构建索引时, 使用查询扩展技术[15]是必要的. 由于代码文本表达和语义的多样性, 仅依靠输入代码的有限特征词汇构建索引或进行检索可能会导致一些相似或相关的结果丢失. 查询扩展通过使用同义词库、word2vec 表达[16]、缩写规则等技术, 找到相似的特征词, 从而扩大特征范围, 进而后续基于索引检索时可以获得更为准确和全面的结果. 在这个过程中, 相似特征词和相关缩写规则的获取需要考虑特定领域词表分析和学习的问题. 具体的处理方式如下.

首先, 将该领域所有代码语料进行预处理, 利用 AST 解析提取出每个代码文件中的所有标识符形成标识符序列. 随后, 对每个标识符进行驼峰式拆分后形成代词序列, 并将所有的单词序列输入 word2vec 中训练得到特定领域的词嵌入模型. 基于该词嵌入模型, 可以计算出词之间的相似度, 例如, "del"和"delete"的相似度会比较高. 再结合一些词形级别的规则, 例如, "del"是"delete"的前缀, 可以获得特定领域的缩写词库, 用于丰富特征词.

基于结构特征构建索引时, 则只需要将三元组作为原子特征, 基于结构特征索引匹配对应克隆组的结构特征数量越多, 意味着代码之间的结构越相似.

2.5 检索推荐流程

在上述特征索引代码数据集建好后, 即可开始进行代码模板推荐操作. 推荐的整个过程主要包括三部分: 输入代码片段的特征识别、基于特征的代码检索和检索结果排序.

2.5.1 输入代码片段的特征识别

模板推荐以代码片段为输入, 因此我们需要从输入的代码中提取特征, 特征提取方法与 2.2 中的相同. 考虑到输入代码可能是短的或不完整的片段, 也可能是相对完整的方法, 我们需要同时开展文本特征和结构特征的提取. 在输入代码可能不完整的情况下, 由于输入内容的结构不完整, 文本特征变得更为重要, 能够传达更多的语义信息.

在实际特征使用过程中, 通过特异性分析学习业务词汇可以获取更具代表性的输入代码的语义信息. 这些语义信息在一定程度上也能反映用户的部分意图. 用户的意图包括对检索业务的意图以及检索代码结构的意图. 这些意图是程序无法预先知道的, 因此需要通过现有代码内容和结构进行推断.

2.5.2 基于特征的代码检索

在获取输入代码特征之后, 基于预先构建好的索引开展代码检索.

2.5.2.1 单一特征检索

检索基于单一文本特征时, 我们需要进行查询扩展, 这也是在构建特征索引时所需的步骤. 借助于扩展后的特征词检索, 能够提高检索的准确度. 对于基于结构特征的检索, 我们可以采用 2.2.2 节中相关于结构特征提取的技术. 通过计算表示结构特征的三元组的向量距离或相似度, 我们可以寻找到拥有相同或相似结构特征的代码. 结构越复杂, 可以描述的结构特征 (即 AST 上的结构相关节点关系) 则越丰富. 这意味着在计算相似性时, 我们能够获取的信息将更加丰富. 相反, 对于简单的结构, 特征信息则较少, 因此更适合进行模糊匹配.

克隆组内的克隆实例通过进行差异性分析并提取了公共部分, 这使我们能够识别出具有相同特性的代码段. 我们可以进一步检查克隆组内所覆盖的方法集, 查询原始代码样例, 并确定重复副本的数量.

值得注意的是, 我们需要为每一个特征克隆组选择一段代表性的代码. 这段代码应该能够归纳出该克隆组的共性, 并能折叠其差异部分. 它可以被视为一个候选的代码模板. 当我们在进行代码检索时, 如果某个特定的克隆组组合通过某些特征匹配了查询, 返回的结果将包含这个代表性的代码. 返回的内容将涵盖共同部分和差异部分 (折叠部分).

在候选的代码模板中, 共性部分由具有共通代码特性的固定代码内容所组成. 然后, 当推荐的代码原始上下文和输入代码的上下文不一致时, 我们可以使用匹配的代码特性, 尤其是结构特性匹配, 来建立代码元素的变量映射. 这种方式使我们能利用到匹配的代码特性, 确定相应的克隆组以及代码模板.

2.5.2.2 联合检索

在实际的代码检索场景中, 根据输入代码的特点, 可能会遇到文本特征和结构特征的丰富程度不同的问题. 单独的文本特征检索有能力在无论代码是否具有明确结构的情况下检索出我们所需的代码, 这使得其适用场景更为广泛. 然而, 这种方法也存在特征包含问题, 也就是说, 当遇到大量具有相同文本特征的代码片段时, 我们很难区分出哪些是我们真正需要的. 在这种情况下, 结构特征就可以发挥它的作用, 作为文本特征的补充. 理想情况下, 文本检索和结构特征检索的结果会有一定的交集. 因此, 我们采用文本特征和结构特征的联合检索, 以提高推荐的准确性和全面性.

在实践中, 我们可以先进行文本特征检索, 再根据检索到的结果进行结构特征检索, 以此过滤出更为准确的代码. 相应地, 结构特征检索也可以作为首要步骤进行, 然后采用文本特征进行进一步的筛选. 我们可以根据输入代码的特点和实际需求, 灵活选择检索的策略和步骤. 可以预见, 这种联合检索方法将更好地满足多元化、复杂化的代码检索需求.

2.5.3 结果排序

在得到文本和结构特征的检索结果后, 我们需要对这些结果进行排序以更好地满足用户需求. 我们首先考虑的是匹配的文本和结构特征的数量. 在特征数量相同的情况下, 我们引入文本权重来作为排序依据. 这个权重是通过 BM25 排名函数计算得来的.

例如, 给定一个查询 Q, 包含文本特征 $f_1,\ldots,f_n$, 代码片段 S 的 *BM25* 分数为:

$$score(S,Q)=\sum_{i=1}^{n} IDF(q_i)\cdot\frac{f(q_i,S)\cdot(k_1+1)}{f(q_i,S)+k_1\cdot\left(1-b+b\cdot\frac{|S|}{avgdl}\right)} \tag{2}$$

其中, $f(q_i,S)$ 是 q_i 在代码片段 S 中出现的次数, $|S|$ 是代码片段文本特征 S 的长度 (以词为单位), $avgdl$ 是从中提取代码片段的文本集合中文本特征的平均长度. k_1 和 b 是自由参数, 通常选择在缺乏高级优化的情况下, $k_1\in[1.2,2.0]$ 和 $b=0.75$. $IDF(q_i)$ 是查询项 q_i 的 IDF (逆文档频率) 权重. 通常计算为:

$$IDF(q_i)=ln\left(\frac{N-n(q_i)+0.5}{n(q_i)+0.5}+1\right) \tag{3}$$

其中, N 是集合中代码片段的总数, $n(q_i)$ 是包含 q_i 的代码片段的数量.

这确保在推荐中考虑了文本和结构特征, 以确保与当前代码片段最匹配的模板在众多推荐结果中排名尽可能高.

3 实验分析

为了评估我们提出的方法, 我们研究了以下三个问题:

• RQ1: 我们能否从代码库中提取足够且高质量的模板以满足实际应用中的需求?

• RQ2: 检索的准确性如何?

• RQ3: 模板推荐在实际开发中的表现如何?

3.1 数据集与评估指标

确保实验具有广泛的代表性对我们至关重要. 因此, 我们从两个关键因素出发来选择项目作为我们的数据集来源: 语言一致性和编码风格一致性. 在选择编程语言方面, 鉴于 Java 语言的广泛使用, 拥有丰富的生态环境和众多的开源项目, 我们将其确定为实验项目的语言选择. 至于编码风格, 为了保证模板推荐的高质量, 我们希望被选项目有着一致的编码风格. 因此选择了遵从阿里巴巴 Java 编码规范的项目作为实验数据集来源.

以这一标准为准, 则从阿里巴巴我们选取了 15 个 Java 开源项目. 这些项目均遵循阿里巴巴 Java 编码规范, 并拥有一致的编码风格. 另外, 我们也随机挑选了 Apache Foundation 和 Google 的 30 个高星项目. 这些项目近期活跃度高, 历史丰富 (至少有 500 次提交记录) 且项目规模大 (包含至少 100 个 Java 文件). 基于这一筛选标准, 我们总计收集了作为实验数据集的 45 个项目.

为了评估我们提出的 CodeSculptor 的有效性, 我们使用了在实验中常用的指标: $Recall@k$ [17]($R@k$). $Recall@k$ 指标评估在前 k 个排名结果中至少包含一个正确结果的查询比例. $Recall@k$ 的计算如下:

$$Recall@k = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\delta(FRank_q \leqslant k) \tag{4}$$

其中, Q 是查询集, $\delta(\cdot)$ 是一个函数, 如果输入为真则返回 1, 否则返回 0. $FRank_q$ 是查询 q 的结果列表中第一个命中结果的排名.

接下来的实验我们会用这些项目去评估 CodeSculptor 在真实场景下的表现. 具体地, 我们会评估 CodeSculptor 基于克隆挖掘出的函数模板荐质量、函数模板检索准确率以及推荐实用性.

3.2 RQ1: 模板分析

CodeSculptor 致力于模板推荐, 旨在帮助开发者从历史代码仓库中重用知识, 以提高编程效率. 然而, 实现这一目标, 模板的质量和数量都是重要影响因素.

质量高是指模板能够有效地代表特定代码段的通用结构和逻辑, 模板本身应当易于理解、使用并且能够适应开发者的具体需求. 这需要模板包含了足够的细节, 同时又保持其通用性, 以适应不同的编程情景. 数量充足的模板库能够提供更多的可能性, 让开发者在解决问题时有更多的选择, 从而提高他们的编程效率.

为了深入探讨此问题, 我们关注以下几个关键维度:

模板数量: 我们必须了解有多少个潜在的模板可以从历史代码中提取. 这有助于评估代码的模块化和可重用性. 大量模板可能意味着代码结构良好、模块化程度高, 从而提高代码的可维护性和可扩展性. 在我们的数据集中, 它包含了 45 个开源项目, 此中我们能够提取 392, 580 个模板. 因此, 显然, 提供的模板数量是充足的.

函数融合度: 我们检查每个模板整合了多少个函数, 以确定模板的复杂性和功能覆盖度. 每个模板中融合的函数数量可以反映其可重用性, 但也可能增加理解和维护代码的难度. 适度的函数融合可以提高代码的可读性和可维护性. 如图 3 所示, 经代码统计模板来源的函数实例数量, 数据集中的大多数模板融合了较少的函数, 特别是在少于 10 个函数的范围内. 然而, 也有一些模板关联了大量的函数, 最多可以覆盖 30 个函数, 有 16 个模板可以达到这个覆盖程度. 这充分证明了一些模板的普适性.

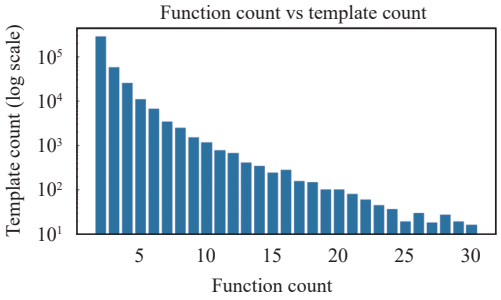

图 3　模板的函数融合度

代码覆盖率: 我们通过计算代码模板与模板来源的原始方法字符数之间的比值得到代码覆盖率, 这个指标可以用于评估模板在原始代码中的覆盖程度. 高覆盖率表明模板与原始代码中的大部分区域存在较高的匹配度, 如果一个模板可以覆盖开发者的实际需求, 那么开发者就可以直接使用这个模板来替代从头开始编写代码, 也有助于减少手动编码过程中可能出现的错误, 因为模板通常都是经过深思熟虑和多次验证的, 其正确性和稳定性通常都能得到保证, 显然这将节省大量的时间和精力. 低覆盖率可能表明低覆盖率可能指示模板的适用范围较小.

在所有的 392, 580 个模板中, 我们计算得出, 模板平均覆盖了原始方法的 60.87%, 这一比例表明如果模板得以恰当推荐, 将在一定程度上减轻开发者的编码负担. 更关键的是, 通过片段级克隆合并形成的模板数量达到了 361, 528, 占模板总量的 92.09%, 这一部分模板是传统的方法无法识别出的, 这一点明确地体现了我们在提取代码模板方面的优越性.

这些分析有助于更全面了解模板的质量和数量, 以及它们在实现实际软件开发中, 如何适应和发挥作用.

3.3 RQ2: 检索准确性

我们设计了消融实验验证不同特征进行检索的效果并随后进行了检索准确性试验. 这两种方法都能够帮助我们更深入地理解我们的方法, 并且评估其效果.

3.3.1 消融实验

为了对比不同特征进行检索的效果, 尤其是结构特征检索对文本特征检索的补充作用, 我们开展了消融实验, 旨在单独评估各个技术决策的影响, 以及它们如何共同作用以提升特征检索的性能. 在我们的实验流程中, 我们观察到, 并非所有的方法都拥有相对明确定义的代码结构. 因此, 我们选取了拥有超过 3 个结构特征的 125, 111 个方法, 这些方法大约占据了所有克隆方法的 36%. 在此基础上, 我们随机选择了 2 115 个测试用例, 这些用例旨在为后续的实验提供评估标准, 具体的测试结果如表 1 所示.

表 1　消融实验结果

	Recall@1	Recall@100
Text Search	99.47%	99.47%
Structure Search	63.80%	99.79%
CodeSculptor	99.84%	100%

通过分析实验结果, 我们发现使用文本特征进行检索的准确度相当高, 几乎所有的预期结果都在返回的结果集中找到了对应. 然而, 当我们只使用结构特征进行检索时, 结果的准确性有所下降. 特别是期望的结构特征出现在结果列表的第一位的可能性相对较低. 然而, 若扩大考察到结果的前 100 位, 我们发现期望的结果通常可以被找到.

因此, 这个实验得出一个重要的结论: 在存在大量文本相似方法的情况下, 文本特征检索可能会受到特征包含的限制, 导致期望的结果并未在返回结果中出现. 然而, 在处理大规模方法数据的情况下, 结构特征通过作为文本特征检索的补充, 有助于提高检索结果的准确性.

3.3.2 准确性实验

CodeSculptor 在经过克隆数据集构建、特征提取和特征索引构建等预处理步骤后, 面临的主要任务是确保推荐内容能反映开发者的检索需求. 具体来说, 当一个方法体包含完整的查询代码片段时, 它应被优先排在检索结果列表的前列. 为了达到此目标, 我们精心地从语料库中现有的方法体中提取出部分代码片段来创建一个微型基准数据集. 对于这些代码片段, 我们期待 CodeSculptor 能将其原始方法体作为检索结果的首项返回. 在我们展开的消融实验中, 总计涉及到 2 115 个方法. 由于需要构造查询代码片段, 因此规定代码行数最少为 12 行, 经过过滤选择了 1917 个方法供实验使用. 针对这个微型基准测试, 我们构建了两类查询代码片段:

1. 连续代码片段: 我们从每个方法体中提取出前 5 行代码, 形成部分查询代码片段.

2. 非连续代码片段: 我们从各个方法体中随机挑选出 5 行代码, 构成部分查询代码片段.

基于此数据集, 我们评估了 CodeSculptor 的检索召回性能, 详细的实验结果如表 2 所示.

表 2 连续与非连续代码片段检索召回率实验结果

	Recall@1	Recall@3	Recall@5	Recall@50	Recall@100
Contiguous	94.00%	96.56%	97.44%	99.47%	99.84%
Non-Contiguous	94.00%	95.67%	96.87%	98.06%	99.11%

根据实验结果, 我们观察到 CodeSculptor 在大多数情况下能准确地把期望结果排在第一位, 无论是针对连续代码片段还是非连续代码片段. 对于非连续代码片段, 虽然其排名稍低于连续代码片段, 但这种差异考虑到连续代码片段的语义连贯性, 是在预期之内的.

3.4 RQ3: 推荐实用性

为了深入探索 CodeSculptor 在实际工作环境中的表现, 我们设计了特殊的案例研究, 目标是评估模板推荐在实际的软件开发任务当中的价值和影响. 本研究使用抽样方法构建了数据集, 以确保所选样本能够统计学上代表全部数据并且最小化抽样误差, 进而满足预定的置信水平要求.

在我们的实验中, 我们旨在实现至少 95% 的置信度. 为此, 我们依据以下置信区间计算公式来确定必要的最小样本量:

$$n = \left(\frac{Z^2 \cdot p \cdot (1-p)}{E^2} \right) \tag{5}$$

其中, n 是所需的样本量, Z 是置信水平对应的 Z 分数 (对于 95% 置信度, Z 值为 1.96), p 是样本中预期比例 (预计的事件发生概率), E 是可接受的误差率. 通过该公式, 确保了抽取的样本能够在统计学上较为准确地反映总体的特性.

通过计算, 本实验在 392, 580 个函数模板的基础上随机选取了 9, 600 的函数模板进行了深入的分析和测试, 以模拟真实世界的软件开发场景, 确保分析结果的置信度至少为 95%. 在 9, 600 个函数模板中, 本实验具体分析了 20 个随机选取的模板, 发现这些模板大致可以被划分为两大类, 亦即连续代码模块和非连续代码模块, 它们各自占据了模板总数的 50%. 具体来看, 连续代码模板包含了进行对象等价性验证、执行 SQL 查询、加载类加载器、将数组转换为字符串等各类功能操作. 而非连续代码模板 (详见代码示例) 则涵盖了 switch 语句、匹配的 try-catch 块、连续的变量检查和相关操作等一个子集. 这些模板都能在不同场景下为代码的重用提供帮助.

代码示例 5: 连续 (左) 与非连续 (右) 代码模板

```
boolean equals(Object o) {                    /* unique code here */
    if (this == o) {                          try
       return true;                           {
    }                                         /* unique code here */
    if (o == null || getClass() != o.getClass()) {   }
```

```
        return false;
    }
    ReadablePathMatcher that = (ReadablePathMatcher) o;
    // Don't use the delegate as toString is unique.
    return Objects.equals(toString, that.toString);
}
```

```
catch(Exception e)
{
fail(e.getMessage());
}
```

为了模拟实际的开发过程, 我们在编辑器中进行代码输入, 然后尝试借助 CodeSculptor 来提升开发效率. 在 9,600 个测试用例中, 我们测量了在连续编程过程中命中预期模板所需输入的字符数量, 以及模板在整个代码中的覆盖率. 测试结果如表 3 所示, 模板的覆盖率在最低的 20% 到最高的 87.2% 之间变化, 同时, 87.2% 的模板都可以通过相对较少的输入字符来命中. 这一结果进一步印证了 CodeSculptor 在实际场景中的高效性.

表 3　案例研究

项目	Java文件名	方法名	字符数量			覆盖率	类型
			输入	模板	方法		
copybara	ReadablePathMatcher	euqals	14	141	311	0.453	1
polardbx-sql	AlterTableGroupBaseTest	initDML1	16	770	1 316	0.585	1
dragonwell8	SecureLoader	getSystemClassLoader	11	295	480	0.615	1
dragonwell8	ComponentColorModel	getAlpha	12	594	1 654	0.359	2
freemarker	ConfigurationTest	testTagSyntaxSetting	25	277	1 371	0.202	2
dragonwell8	SecureLoader	getClassClassLoader	11	268	466	0.575	1
polardbx-sql	AlterTableTest	testAlterTableOneGrOneAtom	45	394	609	0.647	1
polardbx-sql	SOLSelectParser	query	14	1 547	2 488	0.622	2
Alink	OptimizerOuterClass	buildPartial	66	841	1 176	0.715	2
polardbx-sql	SQLExprParser	parseRowFormat	23	1 434	1840	0.779	2
polardbx -sql	SQLStatementParser	parsePurge	23	865	1 611	0.537	2
druid	OracleCursorExpr	equals	18	308	511	0.603	1
dragonwell8	DocLint	init	19	835	2 149	0.389	2
polardbx-sql	SubQueryExecCorresTest	testScalarNormalEquals	34	760	1 279	0.594	1
arthas	ObjectUtils	nullSafeToString	11	317	595	0.533	1
skywalking	BanyanDBNetAddrDAO	BanyanDBNet AddrDAO	24	108	193	0.559	2
guava	TreeMultiset	add	7	730	1 520	0.480	2
dragonwell8	AppendStringBuffer	generateTestBuffer	12	226	385	0.587	1
polardbx-sql	AlterTableMergePar JobFac	merAndMoveToExistTableGro	16	3 183	4 112	0.774	1
logging-log4j2	CompositeFilter	filter	6	188	446	0.422	2

4 用户研究

为深入评估 CodeSculptor 在提升编程效率上的实际价值, 我们尝试复现 Facebook AROMA 实验的“轻量级搜索”阶段. 该阶段如论文和开源代码所示, 要求对代码片段执行 AST 解析, 但此方法对那些无法正确编译和解析的代码片段无效. 正因为如此, 无法将 CodeSculptor 与 AROMA 进行直接的比较. 因此, 我们实施了一项用户研究, 专注于比较开发人员使用或不使用 CodeSculptor 工具时, 完成基于复用的编程任务的效率. 这样的研究有助于我们更好地理解 CodeSculptor 在现实开发中的应用效益.

4.1 研究设置

在本研究中, 我们通过设计一系列编程任务来考察 CodeSculptor 工具在日常软件开发中的应用效率和实用性. 本实验要求参与者使用 CodeSculptor 所推荐的模板来实现特定的开发功能, 并将参与者随机分配至使用 CodeSculptor 的实验组和不使用 CodeSculptor 的对照组, 以便在相同的开发任务下比较两组的性能差异. 为了确

保所有参与者对 CodeSculptor 的操作均有充分了解, 我们在实验开始前对他们进行了详细的工具培训.

实验对象为复旦大学软件工程相关专业的 16 位研究生, 他们均具有 2 至 5 年的 Java 开发经验. 在实验启动前, 我们通过预调查详细了解了参与者的编程技能和历史经验, 并据此将参与者以经验和能力为基础进行配对分组, 随后随机分配至实验组 G1(参与者 P1-P8) 和对照组 G2(参与者 P9-P16).G1 组在执行编程任务时使用 CodeSculptor 工具, 而 G2 组则采用他们熟悉的 IntelliJ IDEA.

考虑到开源项目可能导致理解成本较高并缺乏研究参与者的即时需求, 实验选择了复旦大学软件工程实验室开发的 CodeWisdom 代码大数据平台作为任务基础. 该平台汇集了代码分析、演化分析等多种工具, 以促进代码开发过程的透明度和可控性. 基于此平台, 我们构建了六个不同难度级别的开发任务, 包括与平台业务紧密相关的具体功能实现, 以及一些更通用的辅助性工具开发任务.

我们为参与者提供了 2 小时的完整任务时间, 并要求记录整个开发过程的屏幕录像, 以方便后续对其工作流程和行为模式进行详尽分析. 完成后, 参与者提交了他们的工作录像与开发成品, 并填写了一份问卷调查, 以便我们收集有关任务完成情况和工具使用体验的反馈. 对于实验的成果分析, 我们不仅比较了两组在完成各级别任务所需的时间, 还评估了他们在不同难度层次上的工作效率, 旨在全面评定 CodeSculptor 的工作性能和实际开发价值. 这次研究设计的目的是为了确立 CodeSculptor 作为一个开发辅助工具的优势, 并识别进一步改进的可能路径.

4.2 实验结果

表 4 和表 5 展示了实验的结果. 经过对比两组实验数据, 我们得出以下结论: CodeSculptor 组 (G1) 在完成任务的效率上整体优于 IntelliJ IDEA 组 (G2). 在所有完成任务的 G1 组参与者中, 尽管 P8 参与者没有完成复杂任务, 导致无法计入总体平均耗时, 但考虑到其他完成全部任务的参与者, G1 组的平均总耗时为 65.78 分钟, 方差为 14.17. 对于 G2 组, P11 和 P16 这两名参与者未能完成复杂任务, 导致他们的数据无法用于计算平均总耗时, 但是仍有有 6 名参与者完成了所有任务, 平均总耗时为 83.29 分钟, 方差为 20.08.

表 4 CodeSculptor 组的开发效率

开发人员	简单任务耗时	复杂任务耗时	总耗时	是否完成
P1	15.25	34.85	50.10	是
P2	18.89	35.20	54.09	是
P3	22.97	25.45	48.42	是
P4	24.67	47.33	72.00	是
P5	32.38	51.68	84.06	是
P6	33.98	50.38	84.36	是
P7	29.45	38.00	67.45	是
P8	45.92	/	/	否
平均值	27.93	40.41	65.78	/
方差	9.08	8.96	14.17	/

表 5 IntelliJ IDEA 组的开发效率

开发人员	简单任务耗时	复杂任务耗时	总耗时	是否完成
P9	13.03	30.27	48.30	是
P10	23.00	46.50	69.50	是
P11	28.97	/	/	否
P12	26.67	58.03	84.70	是
P13	36.38	72.68	109.06	是
P14	33.98	69.51	103.49	是
P15	29.25	69.47	98.72	是
P16	35.92	/	/	否
平均值	28.40	57.74	84.80	/
方差	7.25	15.17	22.69	/

在本研究的统计分析中, 我们进一步观察到, 在简单编程功能的实现上, CodeSculptor 工具的用户并未展现出与 IntelliJ IDEA 用户相比的显著优势. 这可能是因为简单功能的实现通常代码量不大, 即使有模板推荐, 效率提升也比较有限. 然而, 在实现更复杂的任务时, 使用 CodeSculptor 的用户表现了更为优异的成绩. 这是因为, IntelliJ IDEA 的用户需要花费更多的时间去理解需求并开始编码以完成满足复杂任务的开发要求. 相比之下, CodeSculptor 工具可以较快展示待复用的代码模板, 使得用户能够更容易理解并通过调整模板完成开发任务. 值得注意的是, 简单任务的完成时间也占据了任务总耗时的很大一部分, 这导致了在总任务时间上, CodeSculptor 用户和 IntelliJ IDEA 用户间的差异被缩小. 尽管如此, CodeSculptor 用户在任务完成上平均仍比 IntelliJ IDEA 用户快了约 19 分钟.

4.3 有效性威胁

我们的用户研究存在三个威胁. 首先, 我们假设两个研究组“相当”, 尽管个体差异总是存在的. 为了解决这个问题, 我们仔细比较了参与者的经验和能力, 并随机将可比较的参与者分配到两个组中. 第二, 我们的培训课程可

能为实验组带来了额外的“热身”经验, 从而导致实验偏差. 根据我们的预研究调查, 所有参与者都熟悉 Eclipse IDE(将 Eclipse 作为他们最常用的 IDE). 因此, 我们认为这样的威胁在很大程度上得到了缓解. 最后, 我们只在一个 Java 设计上进行了受控实验, 因为实验会话有时间限制. 需要进一步的研究来推广我们的发现到更多的系统上.

我们的用户研究面临三个主要威胁. 首先, 虽然我们默认两个研究组是“等效”的, 但实际上个体间总会存在差异. 为了应对这一挑战, 我们对参与者的经验和能力进行了细致的比较, 并将能力相当的参与者随机分配到两个小组中. 其次, 我们举行的培训课程可能为实验组参与者提供了额外的“预热”经验, 这可能引入了实验偏误. 但根据我们预先进行的调查, 所有参与者对 IntelliJ IDEA 都不陌生 (并将 IntelliJ IDEA 视为最常用的 IDE). 因此, 我们认为这种可能的偏差已在很大程度上得到了减轻. 最后, 由于时间限制, 我们的受控实验仅在一个 Java 项目上执行. 未来需要更多研究来验证我们的发现是否能够适用于更广泛的系统环境.

5 相关工作

5.1 代码检索工具

在软件工程领域, 多个工具已经被开发出来[18][19], 使得开发者可以使用部分代码片段作为输入, 来从代码库中检索相关的代码. FaCoY 和 Aroma 便是其中两个极具代表性的工具.

FaCoY 是一个寻找与用户输入代码语义相似的代码片段的工具. 不同于传统的代码检索引擎, FaCoY 实现了一种独特的查询交替策略, 能够辨识出可能与输入代码的功能行为相关的标识符. 它超越了传统的代码检索引擎的局限, 能够检测出 Type-4 的语义代码克隆. 然而, 在语义层面上, FaCoY 尚未尝试从检索结果中创建简洁的推荐.

Aroma 专注于结构化的代码检索, 在从大型代码库中有效推荐简洁代码片段方面表现出色. 而它通过对多样化的查询和编程语言的评估, 结合用户研究, 得以展示其实用性. 但 Aroma 也存在一些问题, 它保留非关键字标识符的完整内容以获得更好的推荐结果, 这可能导致特征维度非常大. 并且, 对于涉及数百万方法的实验数据, 特征数量甚至超过了 3 700 万. 此外, 虽然借助矩阵运算可以提高性能, 然而它在更庞大的代码集上的效果及有效性仍然值得关注.

此外, Aroma 对推荐结果的解释和对现有代码库中实际案例的分析和注释结果却相对较少. 因此, 我们需要更丰富的特征信息 (比如代码在代码库中的出现频率) 以及注解数据 (比如错误处理的完整性) 来解释和排序推荐结果.

另一方面, Senatus 则在 Aroma 的基础上进行了改进. 它通过引入了 De-Skew LSH 算法, 解决了可扩展性和检索质量问题, 显著提升了推荐质量和查询速度. 然而, 即便如此, 与 Aroma 相关的一些问题仍然存在.

5.2 基于模式的代码补全工具

在实现代码补全[20]方面, 有多种方法可供选择, 其中一些主要的策略包括提取代码的结构上下文以及分析最近的编辑历史[21][22][23]. 例如, 学习工具 GraPacc[7]利用 GrouMiner[8]来挖掘以图形表示的编码模式. 然后, GraPacc 会根据输入的代码片段去检索其数据库中匹配的模式, 进而给出代码补全建议. 这种方式被称为基于模式的代码补全.

近年来的一些研究则转向了通过预测代码修改后的下一个 API 调用来增强代码补全的功能[23]. 这种方法利用了代码中的调用序列和编辑历史, 通过预测下一个最可能被调用的 API 来提供补全建议.

然而, 相较于 CodeSculptor, 这些方法的模板数量相对较小, 模板检索的成本可能较高. 由于这些限制, 这些方法在大规模代码库的上下文中可能不具有足够的有效性. 以 GrouMiner 为例, 它最初的设计目标是针对单个项目, 而非广泛的代码库. 因此, 我们认为 CodeSculptor 的优势在于其能适应大规模代码库的环境, 提供大量高质量的模板, 提高代码补全的有效性和准确性.

5.3 克隆检测器

克隆检测工具[24], 例如 SourcererCC[25]、Nicad[26]、SAGA[11]等, 是另一种可供选择的检索技术. 这类工具可以

有效地检测出相似的代码片段来进行推荐. 比如, SAGA 采用了高效的后缀数组设计, 并采用了 GPU 优化技术, 能在 11 分钟内完成对 1 亿行代码的克隆检测. 然而, 这类工具通常只是获取克隆代码的原始数据, 无法直接用于支撑开发任务, 当开发者需要更多的上下文信息来完成他们的开发任务时, 这类工具无法满足需求.

另一方面, 还存在一些与克隆检测相关但提供了更多功能的工具. 例如, CloneDifferentiator[27], 其能精准地识别克隆代码的上下文差异, 使得开发者可以制定精准的查询策略来提取对特定重构任务有用的克隆代码候选.CloneDifferentiator 将克隆代码的上下文表示为程序依赖图, 并通过图差分技术来识别所需的克隆代码上下文差异. 但是, 由于其运行成本较高, CloneDifferentiator 在大规模数据处理上的效率并不理想.

6 总 结

本研究提出了一种新颖的基于克隆的代码模板提取和推荐方法, 名为 CodeSculptor. 这种方法通过在整个代码库中检测代码克隆及提取包含功能差异的方法中的共同上下文, 从而形成开发模板. 进一步地, 我们采用基于文本特征以及三元组格式的结构特征构建一个倒排索引用于推荐和排序. 我们对 45 个精选的开源项目进行的实证研究显示, CodeSculptor 作为一种有效的代码重用工具, 可以显著提升开发效率及代码质量.

在后续的研究中, 我们计划探索将模板进一步抽象化, 对模板的共性部分和可变部分进行更深层次分析, 从而识别和利用在代码编写中反复出现的缺陷模式. 引入这种抽象将有助于进一步减少无意义模板的生成, 同时促进那些可以显著提升代码质量的模板的发掘. 我们还将计划支持模板定制功能, 以增强模板的多样性, 并计划引入更多的代码特性, 使 CodeSculptor 的检索精度提升, 使其更加健壮, 以应对各种开发需求..

References:

[1] Luan SF, 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): 152. [doi: 10.1145/3360578]

[2] Lopes CV, Maj P, Martins P, Saini V, Yang D, Zitny J, Sajnani H, Vitek J. DéjàVu: A map of code duplicates on GitHub. Proceedings of the ACM on Programming Languages, 2017, 1(OOPSLA): 84. [doi: 10.1145/3133908]

[3] ~Syriani E, Luhunu L, Sahraoui H. Systematic mapping study of template-based code generation. Computer Languages, Systems & Structures, 2018, 52: 43–62. [doi: 10.1016/j.cl.2017.11.003]

[4] Wei M, Zhang LQ. Research progress of code search methods. Application Research of Computers, 2021, 38(11): 3215–3221, 3230 (in Chinese with English abstract). [doi: 10.19734/j.issn.1001-3695.2021.04.0096]

[5] Kim K, Kim D, Bissyandé TF, Choi E, Li L, Klein J, Le Traon Y. FaCoY – a code-to-code search engine. In: Proc. of the 2018 IEEE/ACM 40th Int'l Conf. on Software Engineering. Gothenburg: IEEE, 2018. 946–957. [doi: 10.1145/3180155.3180187]

[6] Silavong F, Moran S, Georgiadis A, Saphal R, Otter R. Senatus - a fast and accurate code-to-code recommendation engine. In: Proc. of the 2022 IEEE/ACM 19th Int'l Conf. on Mining Software Repositories. Pittsburgh: IEEE, 2022. 511–523. [doi: 10.1145/3524842.3527947]

[7] Nguyen AT, Nguyen TT, Nguyen HA, Tamrawi A, Nguyen HV, Al-Kofahi J, Nguyen TN. Graph-based pattern-oriented, context-sensitive source code completion. In: Proc. of the 34th Int'l Conf. on Software Engineering (ICSE). Zurich: IEEE, 2012. 69–79. [doi: 10.1109/ICSE.2012.6227205]

[8] Nguyen TT, Nguyen HA, Pham NH, Al-Kofahi JM, Nguyen T N. Graph-based mining of multiple object usage patterns. In: Proc. of the 7th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering. Amsterdam: Association for Computing Machinery, 2009. 383–392. [doi: 10.1145/1595696.1595767]

[9] Wang PC, Svajlenko J, Wu YZ, Xu Y, Roy CK. CCAligner: A token based large-gap clone detector. In: Proc. of the 40th Int'l Conf. on Software Engineering. Gothenburg: IEEE, 2018. 1066–1077. [doi: 10.1145/3180155.3180179]

[10] Robertson S, Zaragoza H. The probabilistic relevance framework: BM25 and beyond. Foundations and Trends® in Information Retrieval, 2009, 3(4): 333–389. [doi: 10.1561/1500000019]

[11] Li GH, Wu YJ, Roy CK, Sun J, Peng X, Zhan NJ, Hu B, Ma JY. SAGA: Efficient and large-scale detection of near-miss clones with GPU acceleration. In: Proc. of the 2020 IEEE 27th Int'l Conf. on Software Analysis, Evolution and Reengineering (SANER). London: IEEE, 2020. 272–283. [doi: 10.1109/SANER48275.2020.9054832]

[12] Grefenstette G. Tokenization. In: Halteren H. Syntactic Wordclass Tagging. Dordrecht: Springer, 1999. 117–133. [doi: 10.1007/978-94-

015-9273-4]

[13] Hirschberg DS. A linear space algorithm for computing maximal common subsequences. Communications of the ACM, 1975, 18(6): 341–343. [doi: 10.1145/360825.360861]

[14] Lin Y, Meng GZ, Xue YX, Xing ZC, Sun J, Peng X, Liu Y, Zhao WY, Dong JS. Mining implicit design templates for actionable code reuse. In: Proc. of the 32nd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Urbana: IEEE, 2017. 394–404. [doi: 10.1109/ASE.2017.8115652]

[15] Lu ML, Sun XB, Wang SW, Lo D, Duan YC. Query expansion via WordNet for effective code search. In: Proc. of the 2015 IEEE 22nd Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER). Montreal: IEEE, 2015. 545–549. [doi: 10.1109/SANER.2015.7081874]

[16] Church KW. Word2Vec. Natural Language Engineering, 2017, 23(1): 155–162. [doi: 10.1017/S1351324916000334]

[17] Buckland M, Gey F. The relationship between recall and precision. Journal of the American Society for Information Science, 1994, 45(1): 12–19. [doi: 10.1002/(SICI)1097-4571(199401)45:1<12::AID-ASI2>3.0.CO;2-L]

[18] Xu Y, Chen XJ, Tang DY, Huang H. Code-search-oriented function multigraph embedding. Journal of Software (in Chinese with English abstract). https://doi.org/10.13328/j.cnki.jos.006940, 2023-07-27/2024-01-20.

[19] Liu ZY, Xing YX, Yu H, Li T, Zhang X D. Retrieval and management technology for industrial-scale massive code. Journal of Software, 2019, 30(5): 1498–1509 (in Chinese with English abstract). [doi: 10.13328/j.cnki.jos.005718]

[20] Hill R, Rideout J. Automatic method completion. In: Proc. of the 19th Int'l Conf. on Automated Software Engineering. Linz: IEEE, 2004. 228–235. [doi: 10.1109/ASE.2004.1342740]

[21] Bruch M, Monperrus M, Mezini M. Learning from examples to improve code completion systems. In: Proc. of the 7th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering. Amsterdam: Association for Computing Machinery, 2009. 213–222. [doi: 10.1145/1595696.1595728]

[22] Robbes R, Lanza M. How program history can improve code completion. In: Proc. of the 23rd IEEE/ACM Int'l Conf. on Automated Software Engineering. L'Aquila: IEEE, 2008. 317–326. [doi: 10.1109/ASE.2008.42]

[23] Holmes R, Murphy GC. Using structural context to recommend source code examples. In: Proc. of the 27th Int'l Conf. on Software Engineering. St. Louis: IEEE, 2005. 117–125. [doi: 10.1109/ICSE.2005.1553554]

[24] Chen QY, Li SP, Yan M, Xia X. Code clone detection: A literature review. Journal of Software, 2019, 30(4): 962–980 (in Chinese with English abstract). [doi: 10.13328/j.cnki.jos.005711]

[25] Sajnani H, Saini V, Svajlenko J, Roy CK, Lopes CV. SourcererCC: Scaling code clone detection to big-code. In: Proc. of the 38th Int'l Conf. on Software Engineering. Austin: IEEE, 2016. 1157–1168. [doi: 10.1145/2884781.2884877]

[26] Cordy JR, Roy CK. The NiCad clone detector. In: Proc. of the 19th Int'l Conf. on Program Comprehension. Kingston: IEEE, 2011. 219–220. [doi: 10.1109/ICPC.2011.26]

[27] Xing ZC, Xue YX, Jarzabek S. Distilling useful clones by contextual differencing. In: Proc. of the 20th Working Conf. on Reverse Engineering (WCRE). Koblenz: IEEE, 2013. 102–111. [doi: 10.1109/WCRE.2013.6671285]

附中文参考文献:

[4] 魏敏, 张丽萍. 代码搜索方法研究进展. 计算机应用研究, 2021, 38(11): 3215–3221, 3230. [doi: 10.19734/j.issn.1001-3695.2021.04.0096]

[18] 徐杨, 陈晓杰, 汤德佑, 黄翰. 面向代码搜索的函数功能多重图嵌入. https://doi.org/10.13328/j.cnki.jos.006940, 2023-07-27/2024-01-20.

[19] 刘志伟, 邢永旭, 于澔, 李涛, 张晓东. 企业级海量代码的检索与管理技术. 软件学报, 2019, 30(5): 1498–1509. [doi: 10.13328/j.cnki.jos.005718]

[24] 陈秋远, 李善平, 鄢萌, 夏鑫. 代码克隆检测研究进展. 软件学报, 2019, 30(4): 962–980. [doi: 10.13328/j.cnki.jos.005711]

肖泉彬(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.

陈源(2000－), 男, 博士生, 主要研究领域为代码克隆和软件成分分析.

吴毅坚(1979－), 男, 博士, 副教授, 博士生导师, CCF 专业会员, 主要研究领域为软件工程, 软件演化分析.

彭鑫(1979－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为软件智能化开发, 云原生与智能化运维, 泛在计算软件系统, 智能网联汽车基础软件.