

中山大学硕士学位论文

题目

Data - Driven Code Commit Evaluation and Comprehension

Auxiliary Methods

学位申请人: _____

指导教师: _____

专业名称: 软件工程

答辩委员会主席(签名): _____

答辩委员会委员(签名): _____

二零一八年四月九日

论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究作出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名:

日期:

学位论文使用授权声明

本人完全了解中山大学有关保留、使用学位论文的规定，即：学校有权保留学位论文并向国家主管部门或其指定机构递交论文的电子版和纸质版，有权将学位论文用于非赢利目的的少量复制并允许论文进入学校图书馆、院系资料室被查阅，有权将学位论文的内容编入有关数据库进行检索，可以采用复印、缩印或其他方法保存学位论文。

学位论文作者签名:

日期: 年 月 日

导师签名:

日期: 年 月 日

论文题目: 题目
专业: 软件工程
硕士生:
指导教师:

摘要

代码提交作为软件演化过程中的重要数据，在代码评审和软件维护中扮演着关键的角色。首先，为了保证代码修改的正确性，评审人员需要对开发人员提交的代码进行评估。另外，随着软件开发人员的更替，新加入的开发人员往往需要从历史的代码提交中理解已有的软件代码变化。然而，由于软件项目的代码质量参差不齐，造成一些代码提交的质量不高，使得开发人员直接理解代码提交的难度增大，从而影响软件维护的效率。因此，本文针对代码提交的评估和理解问题展开研究，并提出了一些解决方案，以提高代码的质量，增强软件的可维护性。

随着开源软件的飞速发展，互联网上积累了数量庞大的代码提交数据。以这些数据为依托，结合机器学习的方法，可以帮助我们更好地评估和理解代码提交。针对代码提交的评估，我们提出了一种数据驱动的代码和注释一致性检测方法。方法中，我们从代码，注释以及代码和注释的关系三种维度中共提取出64种可判别特征，采用机器学习的方法进行模型构建和评估。实验结果表明，我们的方法对不一致的注释检测准确率达到了77.2%，召回率达到了74.6%，且我们的方法可有效帮助开发人员寻找代码提交中与代码不一致的注释。针对代码提交的理解，我们提出了一种基于关键类判定的代码提交理解辅助方法。该方法将关键类判别作为一个二分类问题，从软件演化过程中产生的海量代码提交中，提取可判别特征来度量类的关键性。实验结果表明，我们的方法判定关键类的综合准确率到了88.4%，且相比于开发和评审人员直接理解代码提交，使用关键类信息提示能够显著提高代码提交理解的效率和正确率。在我们的代码修改分析和注释检查系统中，整合了以上两种方法，通过多样的可视化方式

帮助开发和维护人员理解代码修改和评估注释质量。

关键词: 代码提交, 代码注释, 一致性, 关键类, 机器学习

Title: Data - Driven Code Commit Evaluation and Comprehension Auxiliary Methods
Major: Software Engineering
Name:
Supervisor:

Abstract

As an important data in the process of software evolution, the code commit plays an crucial role in code review and software maintenance. On the one hand, in order to ensure the correctness of code modification, reviewers need to evaluate the code commits submitted by developers. On the other hand, with the replacement of software developers, new developers often need to understand existing software code changes from historical code commits. However, because of the uneven quality of the code in software projects, the quality of some code commits is not high, which is difficult for developers to understand the code commits directly and maintain software efficiently. Therefore, this paper studied the evaluation and comprehension of the code commit, and proposed some solutions to improve the quality of code and enhance the maintainability of software.

With the rapid development of open source software, the Internet has accumulated a huge amount of code commit data. Based on these data and combining with machine learning methods, we can evaluate and understand code commits better. In this paper, we proposed a data driven consistency detection method for code and comments to evaluate the quality of code commits. We utilized 64 features, taking the code before and after changes, comments and the relationship between the code and comments into account. Experimental results show that 74.6% of outdated comments can be detected using our method, and 77.2% of our detected outdated comments are real comments which require to be updated. In addition, experimental results indicated that our model can help developers to discover outdated comments in historical versions of existing projects. Aiming at the comprehension of code commits, we proposed a code commit understanding auxiliary method based on core class determination. This method taken the core class discrimination as a binary classification problem, and extracted fea-

tures from the code commits generated in the software evolution process to measure the importance of the class. Multiple datasets of experimental results showed that our method's accuracy reached 88.4%, and compared to the developers understand commits directly, using the core class information as an aid to understand commits can significantly increase the efficiency and accuracy of developers. In our code change analyzing and comments checking system, we integrated the above two methods to help developers understand the code modifications and evaluate the comments quality through various visualization methods.

Keywords: Code Commit, Code Comment, Consistency, Core Class , Machine Learning

目 录

摘要.....	I
Abstract.....	III
目录.....	V
第 1 章 综述	1
1.1 论文研究背景与意义	1
1.2 国内外研究现状	4
1.3 本文的研究内容与主要贡献点	9
1.4 本文的论文结构与章节安排	10
1.5 本章小结	11
第 2 章 基于历史修改模式的影响分析辅助方法	12
2.1 基于历史修改模式的影响分析辅助方法概述	12
2.2 提交语料库构建	13
2.3 文本预处理	14
2.4 软件变化数据提取	16
2.5 多维度的可判别特征选择	21
2.6 一致性检测实验设置与结果评估	29
2.7 本章小结	37
第 3 章 基于关键类判定的代码提交理解辅助方法	39
3.1 关键类判定中的基本概念	39
3.2 关键类判定模型的特征提取	40
3.3 机器学习算法选择及样本优化处理	45
3.4 关键类判定实验设置与结果评估	46
3.5 本章小结	55
第 4 章 代码修改分析与注释检查系统	57
4.1 代码修改分析模块	57
4.2 注释检查模块	60
4.3 基础服务模块	62
4.4 本章小结	62

第 5 章 总结与展望	68
5.1 工作总结	68
5.2 研究展望	69
参考文献	71
致谢	79

第1章 综述

随着互联网与计算机的发展，已经产生了数以亿万计的软件项目。以在开源软件库Github^①上托管的项目为例，2018年官方统计^②数量已超过9600000个，且增速达到40%。海量的软件项目也引出了繁重的软件维护工作，对开发人员提出了巨大的挑战。因此，如何辅助开发人员在软件维护过程更高效的完成代码修改工作，成为软件维护领域的研究热点。开发人员在修改代码过程中，会对软件中其他代码实体产生潜在的影响，必需对这些代码做相应的修改，从而提高了开发人员维护工作的难度和成本。同时，在代码修改完成后，通常需要经过代码审查人员的多次审查，才能最终完成代码修改任务。因此，分析代码修改会产生影响范围以及预估代码修改的完成周期对提高软件维护的效率有关键作用。大数据背景下，如何从软件维护过程中产生的数据里挖掘出有用信息，用于辅助后续的代码修改任务，成为非常关键的研究问题。在Github等版本控制系统中，存在海量软件项目维护过程中产生的数据信息，代码修改信息以提交(Commit)的形式保存。另外，在Gerrit^③代码审核软件中也存在大量代码修改的审核信息。因此，本文针对代码修改的影响分析以及代码修改的完成周期进行深入研究，通过挖掘软件过程中的历史数据，辅助代码修改影响分析和预测代码修改的完成周期，提高代码修改任务的效率和质量。

1.1 论文研究背景与意义

软件的可维护性和可修改性是软件固有的重要特性。软件维护是整个软件生命周期中最关键的一环，占据着70%以上的比重，其主要任务是迎合市场和用户的新需求、修复软件运行过程中的存在的错误、以及对软件性能的优化。软件维护工作被认为是软件生命周期中，最困难和最费人力的工作[1]。软件维护过程中的核心是代码修改工作，由于软件系统的整体性以及系统中各部件的相互依赖关系，代码修改将不可避免地对修改以外的部分产生影响，从而影响

^① Github, <https://github.com>

^② Github, <https://octoverse.github.com>

^③ Gerrit, <https://www.gerritcodereview.com/>

软件的稳定性。为了预估软件维护和代码修改过程的影响范围和程度，修改影响分析成为代码修改任务中的重要一步[2]。另外，代码修改任务的最后一步是代码审核，通常代码修改需要经过多次“修改—审核—再修改”的环节才能最终完成，审核轮次对代码修改的完成周期有直接的影响，提前对代码修改任务的完成周期进行预估，能有效降低软件维护工作的成本。随着软件系统变得越来越庞大和复杂，修改影响分析和修改完成周期的预测是软件维护过程中提高维护效率和质量的有效方法。

开发人员面对的大多数软件修改工作在本项目或者其他项目的历史维护过程中都存在相似的工作，这些历史工作为开发人员完成相似工作时提供了借鉴作用。挖掘历史维护信息，可以辅助相似软件修改工作的影响分析和完成周期预测。大数据背景下，通过挖掘历史数据中的有效信息，再根据历史信息对研究对象进行分析的研究方式得到了广泛的应用，而且研究结果往往有较高的适用性和准确性[3, 4, 5]。在ICSE、ASE 和ICSM 等计算机软件工程顶级会议中，关于数据挖掘在软件工程中应用的研究也吸引了广泛的关注。软件仓库中记录着软件演化的完整历史数据，包括：程序运行数据，缺陷跟踪数据，历史代码修改数据，代码审查数据。这些数据可用于挖掘重要信息，例如项目如何演变[6, 7]，开发人员如何合作[8, 9]，代码修改可能影响的范围[10, 11]等。如图1-1所示，我们总结了在软件工程研究问题中运用数据挖掘方法的总体流程。软件维护和演化过程中存在大量软件工程过程中数据，特别是当前，开源项目广泛托管在Github等版本控制系统中、开源代码审核软件也应用更加广泛，使得软件维护和演化的相关数据获取更加便利。同时，近年来，机器学习取得飞速的发展，在各个领域的应用，都起到了对研究分析的推动作用。在软件工程领域，结合机器学习方法的研究工作，也取得了丰硕的成果。在本文，我们利用机器学习方法，研究开源项目中的代码修改数据和代码审核数据对代码修改影响分析以及对代码修改的完成周期预测的辅助作用。

修改影响分析领域已经有长达30年的研究历史，但是主流的研究方法更加关注系统中代码实体之间的耦合关系以及代码运行信息，以此来分析可能受影响的范围。我们研究发现，大量代码修改工作诸如需求变更、缺陷修复等，都能在软件存储库（本项目或其他项目）中找到相似的代码修改任务。相似的代码修改任务中的代码修改范围对于开发人员确认修改影响范围有直接的借鉴作

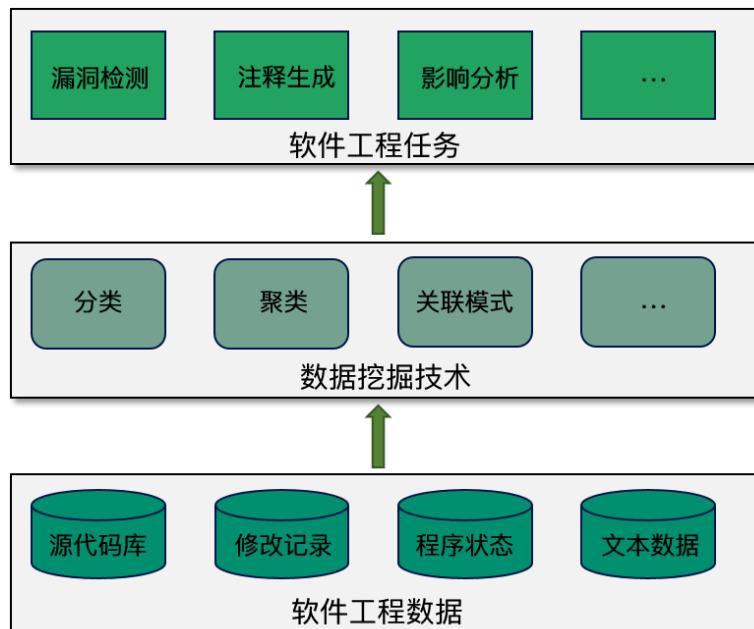


图 1-1 软件工程数据挖掘流程

用。开源项目在版本控制系统中的维护和演化，都是通过代码提交（Commit）的形式进行的。代码提交中的数据包括（如图1-2所示^①）：（1）提交的注释文本，（2）修改前后的代码版本，（3）修改涉及范围，（4）提交编号，（5）提交作者和时间。这些数据包含软件演化过程中修复和改进信息，对后续维护工作起着重要作用[12]。代码修改数据在多个软件工程的研究领域都起到了辅助作用，例如：代码审核评论的自动生成[13, 14]，缺陷预测[15, 16]，修改影响分析[10, 11]等。其中，修改影响分析有助于开发人员在修改代码时，预估可能影响的其他代码实体，辅助开发人员更加高效的完成代码修改工作。本文提出一种基于挖掘代码提交（Commit）信息中的修改模式来辅助修改影响分析的方法，通过关键类判定方法，将提交中的关键类等价为当前修改类，利用关键类的修改影响范围辅助分析当前修改的影响范围。

软件修改工作完成的标志是开发人员的修改提交通过审查人员的审查，我们的调研表明，代码修改的完成周期与修改提交的审查轮数成正相关，预估代码提交的审查轮数能反应代码修改的完成周期。软件代码审查，即让第三方人员对软件系统的代码修改进行审核，是开源和专有软件领域中公认的最佳实践[17, 18]。研究表明，正式代码审核往往能提高软件维护的质量，延长软件生命周期。正式的代码检查流程要求严格的审核标准以确保审核质量的基本水

^① Spring Boot, <https://github.com/spring-projects/spring-boot/commit/2018>

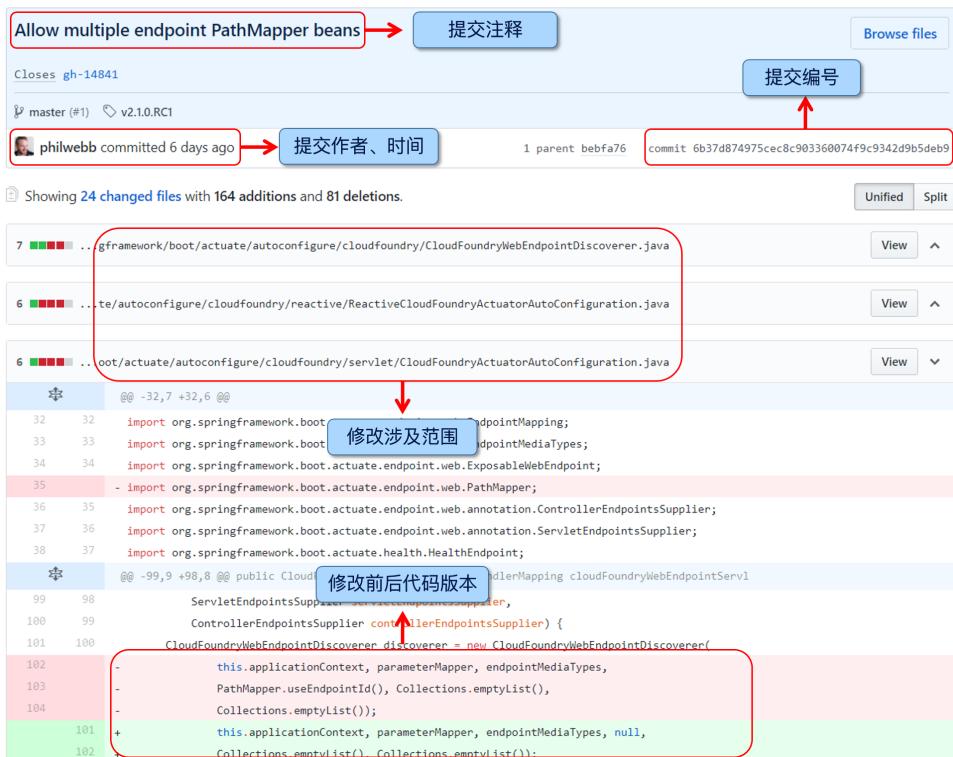


图 1-2 代码提交示例

平[19]。在过去的研中，研究人员已经开发了许多工具和系统来管理软件生命周期的各个方面。对于开发人员而言，软件维护工作中的主要关注点是“如何最大限度的使代码修改提交通过审核人员审核，并最大限度地缩短修改完成时间”。原则上，代码审查是一个透明的过程，旨在评估修改代码的质量。但是，代码审查的执行过程可能受到各种因素的影响，包括技术因素如代码修改难度，也包括非技术因素如项目复杂度，开发人员和审核人员数量等，这些因素很大程度上影响着审查时间和修改的完成时间。然而，很少研究关注代码审查过程中各因素对代码修改完成周期的影响。随着Gerrit等开源代码审查工具的出现，使得代码审查数据（如图1-3所示^①）容易被收集，分析和使用。在本文中，我们从Gerrit中收集代码修改的审查数据，提取有效特征，用于预估代码修改的完成周期。

1.2 国内外研究现状

影响分析领域已有多年的研究历史，许多学者通过不同的方法对此问题进

^① OPEN DAYLIGHT, <https://git.opendaylight.org/gerrit/#/c/62848/1,2018>



图 1-3 代码审核示例

行研究，积累了大量研究成果。此外，得益于机器学习方法的飞速发展以及开源代码托管和审查工具的广泛使用，数据挖掘在软件维护中的应用也受到更多研究人员的关注。本文将从修改影响分析，代码修改模式挖掘，软件存储库挖掘和的代码修改完成周期预测四个方面介绍国内外研究现状以及与本文研究相关的工作。

1.2.1 修改影响分析

修改影响分析可以帮助开发人员理解代码修改，预测修改的影响范围和修改的潜在代价。代码修改往往会导致系统中各部件的关联性而相互波及，如果没有对受波及的部件做相应的调整，会造成各部件之间程序的不一致性[20, 21]。修改影响分析的目的是提前预估代码修改可能造成的影响范围和程度。Bohner等人[22]将影响分析定义为评估软件变更中所有要修改代码的工作。近年来，对于修改影响分析的研究工作不断增多，研究人员提出了许多影响分析的研究方法和支持工具(如图1-4)。

修改影响分析方法主要分为静态影响分析[23, 24, 25]和动态影响分析[26, 27, 28, 29]。静态影响分析通过分析代码之间的语法、语义信息，获取软件部件之间依赖关系，计算修改影响范围。软件项目中某个代码片段的修改通过代码间的依赖关系，可能会传播给其他代码片段，因此分析代码修改的传播机制是静态分析的关键。Breech等人[30]总结了代码修改中的传播机制，并基于这些传播机制建立影响传播图作为中间表示，通过传播图分析影响范围，使得静态分析的精确度得到很大提高。动态影响分析使用特定测试样例，收集程序运行过程

^① JRipples, <http://jripples.sourceforge.net>

^② ImpactMiner, <http://www.cs.wm.edu/semeru/ImpactMiner>

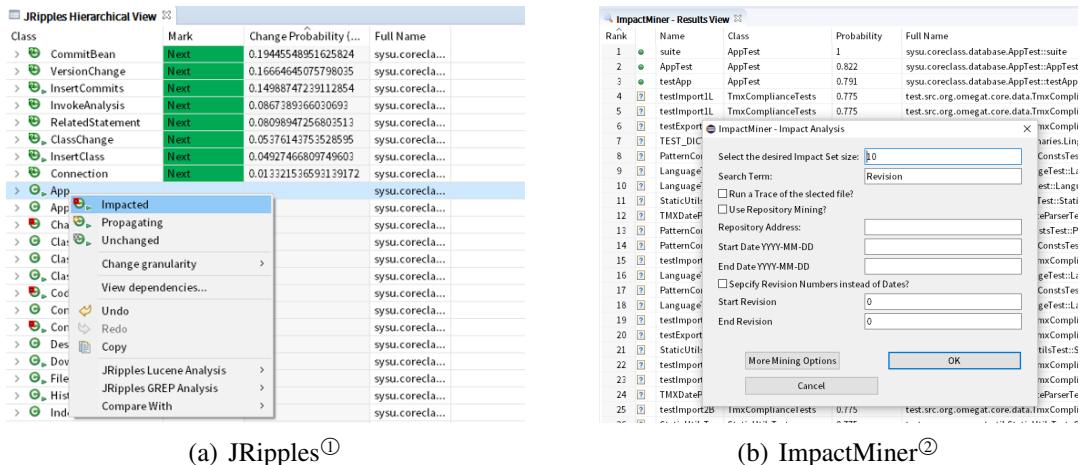


图 1-4 修改影响分析工具

中的数据信息（执行轨迹信息，数据流信息，控制流信息等）计算影响范围。Law 等人[31]提出的PathImpact方法通过收集程序运行时的轨迹信息，构建程序运行路径图，对路径图进行遍历得到修改影响范围。PathImpact方法也是动态分析中使用最广且精度较高的方法。总的来说，静态分析方法由于依赖关系复杂，得到影响范围过大，使得影响分析精度较低；而动态分析方法必须在测试样例运行结束后，才能收集程序运行信息，造成成本过高。针对这个问题，Cai 等人[32]提出了DIVER方法，在PathImpact基础上引入静态依赖图，对程序运行轨迹进行修剪，在较低的成本下得到更高精度影响分析结果。

根据使用技术的不同，修改影响分析还可以划分为基于静态语法依赖关系的影响分析[33, 34]，基于耦合关系的影响分析[35, 36]，以及基于软件存储库挖掘的影响分析[37, 38]等。基于静态语法依赖关系的影响分析是最常用的影响分析技术，通常在代码层次建立依赖图作为中间件，将依赖图中各代码实体之间的可达性作为影响传播的依据。代码层次获得的信息更加丰富，分析结果也更加精确。基于耦合关系的影响分析，计算代码实体之间的结构耦合、概念耦合等，通过耦合关系的强弱计算修改影响范围。基于软件存储库挖掘的影响分析则是根据软件存储库中的历史修改信息，挖掘历史修改中的修改影响范围，将历史修改的影响范围作为当前修改的影响范围。

Li等人[39]的研究表明，基于挖掘软件存储库的影响分析技术越来越受关注。挖掘软件存储库可以发现软件各部件之间重要的历史依赖关系，例如类之间，函数之间或者文档文件。软件维护者可以使用历史依赖关系分析历史变更

中，修改影响是如何传播的，而不是仅依赖于传统的静态或动态代码依赖关系。例如，对一段将数据写入文件的代码进行修改，可能需要对从这个文件读取数据的代码进行相应的调整，但是由于两段代码之间不存在数据和控制流信息，传统静态和动态分析方法将无法捕获这些重要的依赖信息。因此，在传统静态或动态分析技术的基础上，挖掘软件存储库是影响分析方法的良好补充。已有的挖掘软件存储库的方法，仅根据当前修改代码实体，挖掘历史修改记录中该代码修改所产生的影响范围，没有涉及具体的修改内容，即对同一代码做不同的修改，通过历史修改记录所获得影响范围是相同的。本文针对这一问题，提出一种新的历史数据挖掘思路。通过计算当前修改需求、修改代码与历史修改提交中修改描述、修改代码的相似度，得到与当前修改相似的修改提交信息，再使用关键类判定方法判断提交中核心修改的类，将关键类等价为当前修改的类，用提交中关键类的修改模式辅助确认当前修改的影响范围。

1.2.2 代码修改完成周期预测

开发人员在完成代码修改后需要经过第三方审查人员的检查，审查人员根据代码修改情况提出修复建议，以便在代码集成之前识别和修复缺陷。很多研究人员发现，代码审查环节中的许多因素会对代码修改的完成周期产生影响。Patanamon等人[40]研究发现查找合适的代码审查人员是代码修改任务中的关键步骤，不合适的审查人员将严重提高代码维护工作的成本。同时，Patanamon等人[40]还提出一种代码审查人员的推荐方法，该方法根据修改代码的文件路径的相似性来推荐合适的代码审查人员，位于相似文件路径中的文件将由类似的代码审查人员进行管理和审查。与Patanamon等人[40]类似，Oleksii等人[41]的实证研究表明开发人员和审查人员的个人因素会很大程度影响代码修改提交的审查通过率。Baysal等人[42]研究发现诸多非技术因素会影响代码修改的审查通过率，包括：代码修改量，修改提交时间，修改需求的优先次序，代码修改人员和代码审查人员等。他们的方法从WebKit^①项目的问题跟踪和代码审查系统中提取信息，验证各因素的重要性。本文使用机器学习算法从开源代码审查软件中提取审查信息，根据代码审查中各元素预测代码修改的完成周期。由于没有与代码审查周期预测直接相关的工作，接下来主要介绍机器学习在软件维护研

^① WebKit, <https://webkit.org/>

究中的应用。

随着机器学习算法的飞速发展，研究人员已经将机器学习应用于软件维护领域的各个研究工作中。Murali等人[43]提出一个贝叶斯框架，从代码语料库中学习程序规范，使用这些规范检测可能存在缺陷的程序行为。该方法主要的观点是将语料库中的所有规范与实现这些规范的程序语法相关联。Mills等人[44]通过二分类方法实现可追踪性链接恢复，能够自动将所有潜在链接集合中的每个链接分类为有效或者无效。Rath等人[45]利用修改提交的相关信息训练分类器，以识别修改提交所针对的修改问题。Karim等人[46]提出从软件度量指标中提取特征并使用支持向量机和随机森林建立模型来预测软件故障的方法，他们将软件度量指标划分为静态代码度量指标和过程度量，从静态代码度量指标中提取代码行数，循环复杂度以及对象耦合等特征；从过程度量中提取源代码历史变化等特征。Shimonaka等人[47]提出利用机器学习方法从源代码中识别自动生成的代码，该方法认为通过朴素贝叶斯和支持向量机模型从源代码中学习代码的语法信息，可以预测代码是否为自动生成。Nguyen等人[48]提出一种自动映射不同编程语言之间API的方法，该方法从不同编程语言的原代码库中学习API的关联关系。Mario等人[49]从项目源代码以来的API中提取特征，使用机器学习算法实现对软件项目的自动分类。

1.2.3 软件存储库挖掘

软件工程中对软件存储库的挖掘已经有很长的历史，开发人员通过软件存储库数据可以理解软件项目的演化历史，从而更好的完成软件维护和更新工作。软件存储库挖掘的一个方向是从版本控制系统中挖掘历史修改信息。历史修改信息可以帮助开发人员了解软件演化进程中的变更模式从而更高效的完成维护工作。代码修改模式挖掘目的是挖掘历史修改信息中代码实体间是否在修改过程中有关联关系。代码修改中最常见的是同步修改模式，例如，如果从历史信息发现代码实体 e_1 总是与代码实体 e_2 同时进行修改，当开发人员再次对 e_1 进行修改时，可预估 e_2 需做相应的修改。根据代码同步修改模式，开发人员可以在面对代码修改需求时，预测需要共同修改的代码实体。Bouktif等人[21]定义了同步修改模式的一般概念，即描述在小时间范围内共同变化的代码实体，并使用模式识别中的动态时间扭曲技术对历史修改数据进行分组，提取相似的修改模式。

Ying等人[50]通过基于频率计数的频繁模式挖掘技术从源代码更改历史中挖掘同步修改的源代码。Zimmermann等人[51]利用历史修改代码中的关联规则挖掘出代码实体的同步修改模式，并实现了一个原型系统ROSE^①，为开发人员预测需共同变化的代码，预防因不完整修改而导致的错误，并且能检测出用传统程序分析方法无法得到的耦合关系。Ajienka等人[52]的研究表明，频繁同步修改的代码实体之间存在很强的耦合关系，挖掘语义耦合关系有助于识别修改模式。除了同步修改模式外，代码修改中还存在许多其他修改模式。Jaafar等人[53]的研究中提出了两种新的代码修改模式，代码异步修改模式和代码移相修改模式。代码异步修改模式指的是在大时间区间内共同修改的代码；代码移相修改模式指的是频繁在相同时间间隔进行修改的代码。Stephane等人[54]通过聚类的方法，对一定时间周期内完成相似修改的代码进行归类，划分不同的修改模式。与Stephane等人[54]类似，Fluri等人[55]利用层次聚类实现了修改模式的半自动挖掘。本文通过挖掘历史修改提交中的修改模式，辅助当前修改的影响分析。

软件存储库挖掘的另一个研究方向是对源代码的挖掘。Michail等人[56]运用数据挖掘技术检测在不同程序中如何复用代码。了解代码的复用模式可以有效减少开发人员工作量。Lin等人[57]利用频繁挖掘技术在源代码中提取编程规则，他们的研究表明，违反编程规则的代码可能存在缺陷。Holmes等人[58]通过挖掘源代码库中代码的结构上下文，向开发人员展示相关API的用法。类似地，Bruch等人[59]提出从代码库中学习从而提升IDE中代码的补全效果。Zaidman等人[60]通过挖掘软件演化数据，研究工程代码和测试代码在软件演化中如何共同发展。

1.3 本文的研究内容与主要贡献点

本文依托海量的代码修改提交数据以及代码修改审查数据，结合机器学习方法，针对代码修改影响分析以及代码修改完成周期预测进行深入研究。本文的主要研究内容如下所述：

(1) 针对代码修改影响分析，本文提出一种基于历史修改模式的影响分析辅助方法。本文从开源项目中获取代码提交历史数据构建提交语料库，通过计算

^① ROSE: <http://www.st.cs.uni-sb.de/softcvo/>

当前修改的自然语言描述与提交注释信息以之间以及当前修改前后代码与提交中修改前后代码之间的相似度匹配最相似的提交。通过关键类判定方法判定提交中的关键类，将关键类等价为当前修改类，以关键类在提交中修改模式指导当前修改的影响分析。最后通过传统影响分析方法获取当前修改的初始影响集，结合18种程序实体间的耦合关系，分析提交中关键类与其他类的耦合关系以及当前修改类与初始影响集中其他类的耦合关系，根据耦合关系的相似度将关键类的修改模式映射回当前修改，对初始影响集中的类进行重排序得到最终修改集。最后，通过实验对比初始影响集与最终影响集的影响分析效果。

(2) 针对修改完成周期的预测，本文提出一种基于可判别性特征的修改完成周期预测方法。本文从开源代码审查软件中获取海量代码审查历史数据，提取可判别特征，以代码修改审查轮次预估代码修改完成周期，并结合机器学习方法建立代码完成周期预测的模型。从代码审查人员以及代码修改人员信息中提取非技术维度的影响特征，从提交注释文本以及修改前后代码中提取技术维度的影响特征，结合多维度的特征训练机器学习模型。

本文的主要创新点如下：

(1) 在修改影响分析方面，本文引入开源项目中的相似修改信息对传统影响分析方法的结果进行优化，效果得到提升。与其他挖掘项目历史修改信息的方法不同的是，本文从多个开源项目中基于修改内容相似度匹配历史提交，而传统方法仅从当前修改类的项目修改历史中检索与当前类共同修改的类集，不涉及具体修改内容。实验结果表明，本文提出的影响分析辅助方法，在多个开源项目上都能提升传统影响分析工具的精确率和召回率。

(2) 在修改完成周期预测方面，本文创新地提出通过挖掘代码审查信息中的可判别特征，预测代码修改周期的方法。实验结果表明，代码审查信息中存在大量影响代码修改周期的因素，本文基于这些因素，结合机器学习模型，完成了对代码修改周期较好的预估结果。

1.4 本文的论文结构与章节安排

本文共分为五章，章节内容安排如下：

第一章主要阐述了代码修改影响分析和代码修改完成周期预测在软件维护

过程中的重要性，概述了代码修改影响分析、代码修改模式挖掘、代码修改周期预测以及软件存储库挖掘的国内外研究现状，以及介绍了本文的研究内容和主要贡献点。

第二章提出了基于历史修改模式的影响分析辅助方法。该章主要介绍代码修改提交语料库的构建、相似提交的筛选、历史修改模式的映射以及影响集的优化等。

第三章提出了基于可判别性特征的修改周期预测方法。该章主要介绍代码审查数据的处理、特征的提取、算法选择以及模型优化和评估等。

第四章的主要内容是介绍本文提出的修改影响分析辅助方法和代码修改周期预测方法的系统设计和实现，分别介绍了系统实现方法、主要功能以及系统展示。

第五章对本文的工作进行总结。主要总结本文提出的方法，并分析这些方法存在的不足与局限，最后，指出在将来的研究工作中如何进行改进。

1.5 本章小结

本章首先介绍了代码修改影响分析和代码修改完成周期预测在软件维护过程中的重要性，并对与本文相关的技术和研究领域进行了概述，分别介绍了代码修改影响分析、代码修改模式挖掘、代码修改周期预测以及软件存储库挖掘的国内外研究现状。最后概述了本文的主要研究内容以及本文的贡献点，并对本文的章节安排作了简要的介绍。

第2章 基于历史修改模式的影响分析辅助方法

对于一个软件系统来说，经过多年的开发历史，系统中各个软件实体之间存在非常复杂关联关系，当开发人员需要对其中一个软件实体进行修改时，必然会影响到其他软件实体，且影响范围会随着实体间的关联关系不断传播。因此，修改影响分析成为软件修改工作中的关键一步。修改影响分析的目的是评估一项修改任务可能带来的风险以及受影响的范围和程度。软件修改的主要目的包括增加新功能、修复缺陷或者适应新的用户需求。在同一项目或不同项目的演化历史中往往存在着相似的修改需求，这些相似修改的修改模式对于当前的修改任务有辅助作用。传统的静态影响分析以及动态影响分析方法难以捕获软件项目中复杂的依赖关系，引入历史修改模式信息可以对传统影响分析的结果进行优化，提升影响分析效果。已有的挖掘历史修改信息的影响分析方法没有涉及具体修改内容，本文从版本控制系统中收集优质开源项目的修改提交数据构建提交语料库，通过修改代码相似度以及修改需求的相似度检索相似的修改提交，再借助关键类判定方法将相似提交中的关键类作为当前修改类的等价类，引入关键类的修改模式对传统影响分析结果进行优化，从而获得最终的影响集。

2.1 基于历史修改模式的影响分析辅助方法概述

图2-1是基于历史修改模式的影响分析辅助方法总览，方法主要分为三个步骤：（1）构建历史提交语料库，（2）检索相似提交，（3）辅助影响分析。第一步，首先从版本控制器中收集大量不同项目下的代码修改提交数据，并将这些提交数据存储于本地仓库中，提交数据包括修改前代码片段、修改后代码片段以及修改注释信息。分别对每一条提交数据进行文本预处理，构建提交语料库，语料库中每一条语料包含相应提交中的代码修改信息和注释信息。第二步，对当前修改工作中的修改需求(文本描述)和修改前后的代码片段做相同的文本预处理；使用词嵌入方法Word2vec训练提交语料库，得到词嵌入模型；利用词嵌入模型计算当前修改与历史提交的向量相似度，得到相似修改提交列表。第三步，

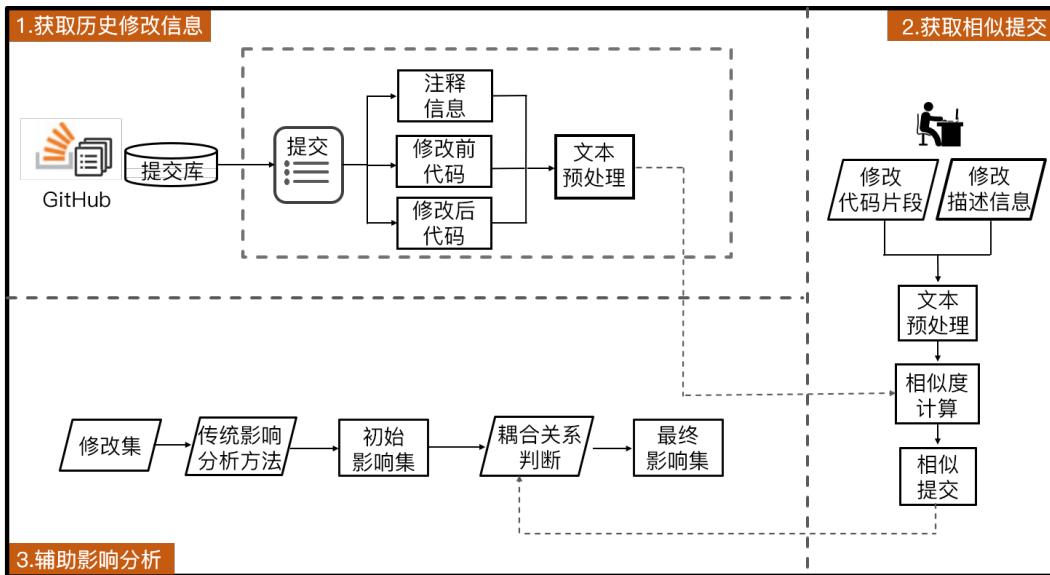


图 2-1 基于历史修改模式的影响分析辅助方法总览

使用关键类判定方法识别相似提交中的关键类，将关键类作为当前修改类的等价类；提取关键类与提交中其他类的耦合关系；使用传统影响分析方法得到当前修改的初始影响集，提取当前修改类与初始影响集中其他类的耦合关系；利用耦合关系的相似度将提交中关键类的修改模式映射回当前修改类，对初始影响集进行优化，得到最终影响集。

2.2 提交语料库构建

本文中用于构建提交语料库的数据来源于版本控制系统Github、Sourceforge中的开源项目，这些开源项目经过长期维护存在大量代码修改的提交数据。我们从版本控制系统中筛选出182个开源项目，并从这些开源项目中收集了94778个提交数据用于构建提交语料库。

2.2.1 提交数据优化

本文用于构建提交语料库及验证的数据来源于开源项目中，而开源项目中代码提交数据质量良莠不齐，为了防止质量较差的提交数据对影响分析产生负面的优化效果，我们需要对用于构建语料库的提交数据进行筛选。我们对大量提交数据进行观察后，发现开源项目中修改提交主要存在以下问题(如图2-2所示)：(1) 修改提交中注释信息缺失或过短(小于3个单词)，这类提交缺乏对修

改内容的有效描述信息，将影响后续相似提交的检索；（2）提交注释信息过长（大于200个单词），这类提交的注释信息中往往罗列了该修改工作中大部门琐碎的修改内容，难以判断其核心修改部分；（3）提交中只涉及一个类的代码修改，这类提交数据由于只包含一个类不存在可借鉴的修改模式；（4）提交中涉及超过二十个类的代码修改，这类提交数据通常是由多个普通提交组合而成，一般出现在版本更新的代码提交中。

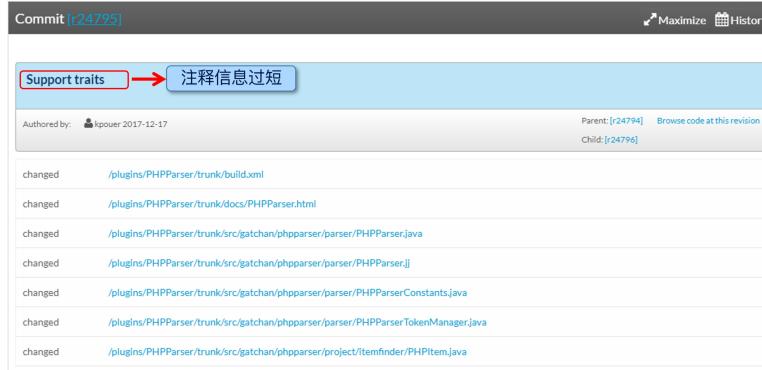
通过对存在问题的提交数据筛选后，我们收集的提交数据包含182个开源项目共94778条。一条提交数据主要包含：提交编号、作者、提交日期、提交注释文本、修改前版本代码和修改后代码版本代码等。在相似提交检索任务中，本文通过修改描述文本的相似度以及代码修改片段的相似度来衡量当前修改与提交的相似度，因此，我们收集的提交数据仅需保留提交注释文本、修改前版本代码和修改后版本代码。

2.3 文本预处理

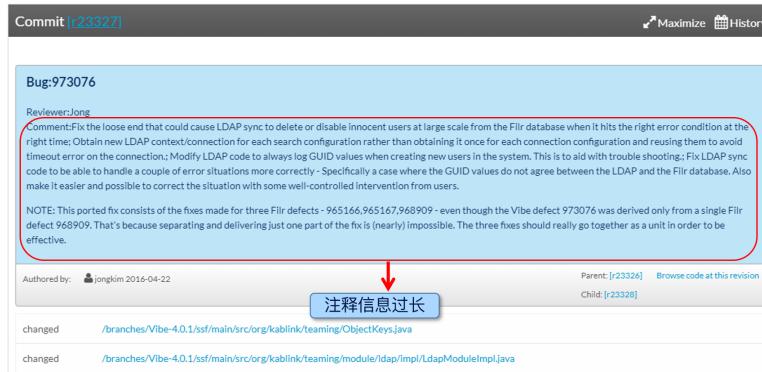
提交中代码数据和注释文本中通常存在许多噪声字符，这些噪声可能会削弱文本中原有的语义信息。从开源项目中收集的提交数据直接用于相似提交检索有可能起负面作用，我们需要对提交中的代码及注释文本进行预处理。

第一步，对代码及注释文本使用相同预处理方法。对代码及注释文本中的标点符号、特殊字符、数字等进行过滤，并通过空格、换行符将代码及注释文本分别转化为一系列字符串和或者单词。将文本中所有的单词同一规范化为小写单词，如“*Text*”转换为“*text*”。另外，开发人员喜欢在编写注释和代码时，使用缩略词，在文本预处理中，需要对缩略词进行补全，如“*Info*”转换为“*information*”。同时，词形还原和词干提取也是文本预处理中重要的一步，通过对单词规范化，可以显著提高文本相似度计算中的精确度。本文借助Wordnet对所有的文本做词形还原和词干提取。词形还原的目的是将不同形式和不同时态的单词还原为一般形式，如符数“*classes*”还原为“*class*”，进行时的“*running*”还原为“*run*”等。词干提取的目的是提取文本中单词的词干或词根表示，如“*effective*”转换为“*effect*”，“*happiness*”转换为“*happy*”。

第二步，由于代码文本与自然语言形式的注释文本之间存在明显区别，



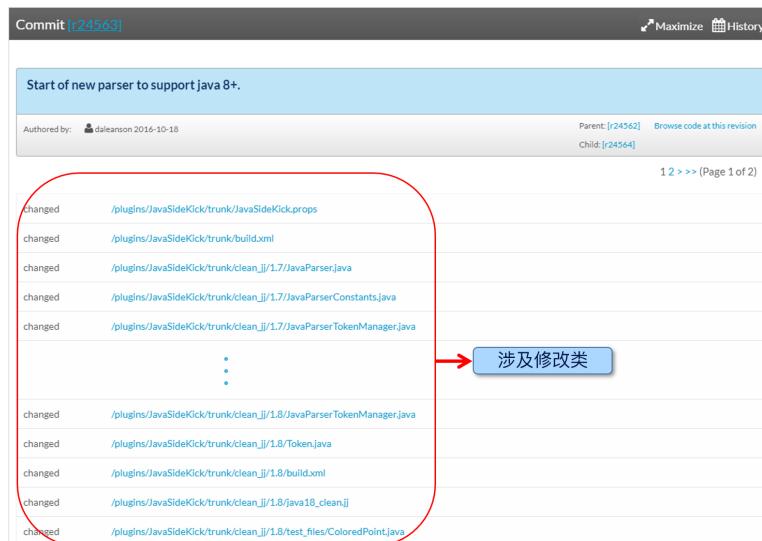
(a) 提交注释信息过短



(b) 提交注释信息过长



(c) 提交只涉及一个类修改



(d) 提交涉及超过二十个类修改

图 2-2 提交涉及超过二十个类修改

需要对代码文本使用额外的预处理方法。语法分析从程序逻辑的角度衡量代码间的相似度，而语义分析是直接根据代码中的标识符判定代码的相似度。但是，代码文本中许多标识符并不能对语义相似度的分析起到促进作用（例如，“*asaa*”，“*a*”，“*b*”等等）。本文通过使用一系列方法过滤代码文本中特定的标识符，包括：(1)过滤代码文本中的虚词，如“*and*”，“*a*”，“*an*”等；(2)过滤不表示单词的字符串，如“*ttt*”，“*hgkk*”等等；(3)将使用驼峰命名法的词汇分割成单独的单词，如“*removeContextInfo*”分割成“*remove*”，“*Context*”和“*Info*”。经过一些列文本预处理后，提交中的原始代码片段仅保留了具有语义信息的实词，每块代码片段相当于一个文本文档。

另外，本方法除了需要对历史提交数据进行文本预处理之外，还需要前影响分析对象进行相同的文本预处理。当前修改需求的自然语言描述相当于提交中的注释文本，当前修改的代码变更片段相当于提交中代码修改片段。

2.4 软件变化数据提取

本节详细介绍了从版本控制库中提取软件变化的过程。我们从版本控制库中提取代码提交中相邻两个版本间的代码和注释，然后按照启发式规则确定注释的作用域。其中，注释作用域是指与注释所关联的代码段的集合。我们把变化前和变化后的注释、注释作用域中的代码片段以及注释作用域中变化的程序语句看作一个软件变化。在进行数据提取时，只关注注释作用域中包含程序语句变化的软件变化，过滤掉不包含程序语句变化的软件变化。

2.4.1 注释类型介绍

在Java语言中，有三种类型的注释，分别为文档注释(Javadoc Comment)，块注释(Block Comment)和行注释(Line Comment)[?]。如图2-3所示^①。其中，图2-3(a)中的第1-10行为方法*indexOf*的文档注释，图2-3(b)中的第4-7行以及第15行为块注释，图2-3(c)中的第6行以及第11-12行为行注释。由于文档注释具有良好的结构，有很多研究学者针对这一类型的注释与代码之间的一致性问题做了充分的研究[?, ?, ?, ?]，且在一致性的判别中已有比较理想的结果。所以，本文主要关注块注释和行注释与代码的一致性问题。另外，在我们的实验过程中发现，

^① JDK/java/lang/String.java,2018

```

1. /**
2. * Returns the index within this string of the first occurrence of
3. * the specified substring, starting at the specified index.
4. *
5. * @param str      the substring to search for.
6. * @param fromIndex the index from which to start the search.
7. * @return the index of the first occurrence of the specified
8. *         substring, starting at the specified index,
9. *         or {@code -1} if there is no such occurrence.
10.*/
11. public int indexOf(String str, int fromIndex) {
12.     return indexOf(value, 0, value.length,
13.                    str.value, 0, str.value.length, fromIndex);
14. }

```

(a) 文档注释示例

```

1. static int lastIndexOf(char[] source, int sourceOffset,
2.                        char[] target, int targetOffset, int targetCount,
3.                        int fromIndex, int sourceCount) {
4. /*
5. * Check arguments; return immediately where possible. For
6. * consistency, don't check for null str.
7. */
8. int rightIndex = sourceCount - targetCount;
9. if (fromIndex < 0) {
10.    return -1;
11. }
12. if (fromIndex > rightIndex) {
13.    fromIndex = rightIndex;
14. }
15. /* Empty string always matches. */
16. if (targetCount == 0) {
17.    return fromIndex;
18. }
19. ...
20. }

```

(b) 块注释示例

```

1. public int indexOf(int ch, int fromIndex) {
2.     final int max = value.length;
3.     if (fromIndex < 0) {
4.         fromIndex = 0;
5.     } else if (fromIndex >= max) {
6.         // Note: fromIndex might be near -1>>>1.
7.         return -1;
8.     }
9.
10.    if (ch < Character.MIN_SUPPLEMENTARY_CODE_POINT) {
11.        // handle most cases here (ch is a BMP code point or a
12.        // negative value (invalid code point))
13.        final char[] value = this.value;
14.        for (int i = fromIndex; i < max; i++) {
15.            if (value[i] == ch) {
16.                return i;
17.            }
18.            ...
19.        }
20.    }
21. }

```

(c) 行注释示例

图 2-3 不同类型的注释示例

开发人员在对代码编写注释时，块注释和行注释的选择界限往往比较模糊。对于单行注释的选择，可以以块注释的形式展现，如图2-3(b)中的第15行所示。也可以以行注释的形式展现，如图2-3(c)中的第6行所示。对于多行注释，图2-3(b)中的第4-7行中使用了块注释，而图2-3(c)中的第11-12行则使用了行注释。因此，我们在实验过程中将块注释和行注释看作同种类型的注释，并将多行的行注释合并为一个注释。

2.4.2 注释作用域检测算法

在对代码和注释的一致性问题研究中，以往研究工作的主要研究对象为文档注释及其相关代码的一致性。由于这类注释具有良好的结构信息，且其关联的程序对象通常为方法成员和属性成员，所以与注释关联的代码范围容易确定。而对于块注释和行注释而言，由于其具有结构松散，使用灵活的特点，确定它们所关联的代码范围往往比较困难。因此，本文提出了一种注释作用域检测的启发式算法，用于确定与注释相关联的代码范围。算法描述如算法1所示。算法1将源代码文件 S ，待检测的注释集合 CM 作为输入；将新生成的注释作用域集合 $CSSet$ 作为输出。首先，使用源代码文件构造抽象语法树(Abstract Syntax Tree,AST)[?]，得到源代码的方法集合 $methodSet$ (第5, 6行)。然后，对于 CM 中的每个注释 $comment$ ，从 $methodSet$ 中得到 $comment$ 所在的方法 $method$ ，并将注释作用域 CS 的结束行 $endLine$ 设为 $method$ 的结束行(第9-11行)。紧接着获取 $method$ 的语句集合 $statementSet$ ，并遍历该语句集合(第12-22行)。遍历时首先寻找到紧邻注释的 $statement$ ，将该 $statement$ 的起始行 $startLine$ 作为 CS 的起始行(第16, 17行)。其次，对于在 CS 作用域范围内的 $statement$ ，将其加入到 CS 的语句集合 $codeSet$ 中(第13-15行)。再次，如果 $statement$ 包含了待检测作用域的注释，且该 $statement$ 的结束行 $endLine$ 小于 CS 当前的结束行，则将 CS 的结束行修改为 $statement$ 的 $endLine$ (第21-23行)。最后，获得与 $comment$ 处于同一语句块中且紧邻的下一个注释 $next_comment$ ，如果 $next_comment$ 的起始行小于 CS 的结束行，则将 CS 的结束行修改为 $next_comment$ 的起始行的前一行(第23-26行)。如果该注释遍历完 $statementSet$ 中的所有语句，则结束一个 $comment$ 的作用域检测，跳出 $methodSet$ 循环(第28行)。当 CM 中的每个 $comment$ 都完成作用域检测后，返回 $CSSet$ 集合(第33行)。

算法1: 注释作用域检测算法

```
1: Input:  $S$ : Source code;  $CM$ : The set of comments;  
2: Output:  $CSSet$ : The set of commentScopes;  
3: CommentScopeExtraction ( $S, CM$ ):  
4:    $CSSet = \{\}$   
5:    $astTree = \text{getASTTree}(S)$   
6:    $methodSet = \text{getMethodSet}(astTree)$   
7:   foreach  $comment \in C$  do  
8:      $CS = (comment, null, 0, 0)$   
9:     foreach  $method \in methodSet$  do  
10:       if  $comment \in method$   
11:          $codeSet = \{\}$ ;  $startLine = 0$ ;  $endLine = method.endLine$   
12:          $statementSet = \text{getStatementSet}(method)$   
13:         foreach  $statement \in statementSet$  do  
14:           if  $statement.startLine > comment.startLine \& statement.endLine < endLine$  do  
15:              $codeSet.add(statement)$   
16:             if  $startLine == 0$  do:  $startLine == statement.startLine$   
17:               end if  
18:             end if  
19:             if  $comment \in statement \&& statement.endLine < endLine$  do  
20:                $endLine = statement.endLine$   
21:             end if  
22:           end foreach  
23:            $next_comment = \text{getNextComment}(comment, C)$   
24:           if  $next_comment.startLine - 1 < endLine$  do  
25:              $endLine = next_comment.startLine - 1$   
26:           end if  
27:            $CS.c = codeSet$ ;  $CS.s = startLine$ ;  $CS.e = endLine$   
28:           goto 7  
29:         end if  
30:       end foreach  
31:        $CSSet.add(CS)$   
32:     end foreach  
33:   return  $CSSet$ 
```

```

public void arcTo(double rx,double ry,
                  double xAxisRotation,boolean largeArcFlag,
                  boolean sweepFlag,double x,double y) {

    //Ensure radii are valid
    if(rx == 0 || ry == 0) {
        lineTo(x,y);
        return;
    }

    // Get the current(x,y) coordinates of the path
    Point2D.Double lastPoint = (Point2D.Double)getCurrentPoint();
    double x0 = lastPoint.getX();
    double y0 = lastPoint.getY();
    if(x0==x&&y0==y) {
        // If the endpoints(x,y) and (x0,y0) are identical, then this
        // if equivalent to omitting the elliptical arc segment entirely.
        return;
    }

    // Compute the half distance between the current and the final point
    double dx2 = (x0-x)/2d;
    double dy2 = (y0-y)/2d;
    // Convert angle from degrees to radians
    double angle = toRadians(xAxisRotation);
    double cosAngle = cos(angle);
    double sinAngle = sin(angle);
}

```

图 2-4 注释作用域检测示例

图2-4为注释作用域划分示例。如图所示，该代码片段包含5个注释，其中，第3个注释包含在第二个注释的作用域内，我们在作出注释作用域检测后，将具有包含关系的注释合并为一个注释。最后，这段代码提取了4个代码-注释对，分别对应图2-4中的标签1至标签4。

2.4.3 软件变化提取

软件变化是指在两个提交版本间的注释和代码以及代码中的程序语句变化的集合，一个软件变化的代码范围实际上是由对应注释的作用域确定的。软件变化提取主要包含以下几步：(1)版本间注释配对。我们从两个版本的源代码中提取出注释信息，并对两个版本的注释进行一一匹配；(2)注释类型过滤。按照注释的类型对注释进行过滤，过滤掉文档注释和单行注释(与代码处在同一行的行注释)；(3)注释作用域检测。按照算法1对注释进行作用域检测；(4)代码变化提取。对两个版本的注释作用域内的代码片段进行比较，提取代码变化，我们丢弃掉不包含代码变化的软件变化，例如两个版本的代码片段完全一致或者只涉及注释变化或代码格式的变化；(5)软件变化提取。根据获得的两个版本的注释信息，注释作用域信息，代码信息以及代码变化信息，进行软件变化的提

取。

```

project:freecol commit_id:5829 class_name:TransportMissionTest.java
//Verify that the outcome of the combat is a return to Europe for repairs
//and also invalidation of the transport mission as side effect
assertTrue(galleon.isUnderRepair());
assertFalse(aiUnit.getMission().isValid());
try {
    // this will call AIPlayer.abortInvalidMissions() and change the carrier mission
    aiPlayer.startWorking();
    assertFalse(aiUnit.getMission() instanceof TransportMission); 2
} finally{ 1
    ...
}

```

Before Change


```

//Verify that the outcome of the combat is a return to Europe for repairs
//and also invalidation of the transport mission as side effect
assertTrue(galleon.isUnderRepair());
assertFalse(aiUnit.getMission().isValid());
} 1
// this will call AIPlayer.abortInvalidMissions() and change the carrier mission
aiPlayer.startWorking();
assertFalse(aiUnit.getMission() instanceof TransportMission); 2
}

```

After Change

图 2-5 删除语句块引起注释作用域缩小示例

```

project:KabLink commit_id:12173 class_name:WorkflowModuleImpl.java
//need to save explicitly - actions called by the node.enter may look it up
getCoreDao().save(ws);
entry.addWorkflowState(ws); 1
//Start the workflow process at the initial state
ExecutionContext executionContext = new ExecutionContext(token);
node.enter(executionContext);
context.save(pi); 2
}


```

Before Change


```

//need to save explicitly - actions called by the node.enter may look it up
getCoreDao().save(ws);
entry.addWorkflowState(ws);
if (!options.containsKey(ObjectKeys.INPUT_OPTION_NO_WORKFLOW) ||
!(Boolean)options.get(ObjectKeys.INPUT_OPTION_NO_WORKFLOW)){ 1
    //Start the workflow process at the initial state
    ExecutionContext executionContext = new ExecutionContext(token);
    node.enter(executionContext);
}
context.save(pi); 2
}


```

After Change

图 2-6 新增语句块引起注释作用域扩大示例

在进行软件变化提取的过程中，由于语句块的引入或删除常造成注释作用域的扩大或缩小，如图2-5所示，在代码变化过程中删除了`try-finally`语句块，且在语句块内包含了注释。在代码变化后第一个注释的作用域(红色框部分)缩小到第二个注释所在位置的前一行，而实际上`try-finally`内的代码并未删除。而在图2-6所示的示例中，可以看到在新版本中增加了`if`语句块(红色字体所示)，将第一个注释的作用域扩大到整段代码的末尾(红色框部分)。在对代码和注释的一致性评估时，这两种情况可能会让机器认为软件变化删除了一大段代码或者新增了一大段代码，从而对代码和注释的一致性作出错误的判断。为了尽量减少这两种情况对我们的一致性评估的影响，我们将对它们进行识别，并将图2-5和图2-6中的两个软件变化(红色框和蓝色框)进行合并，将它们看成是一个软件变化。

2.5 多维度的可判别特征选择

本节详细介绍了从软件变化中选取特征的过程。代码和注释的一致性可以由代码和代码的变化，注释的文本信息以及代码和注释的相关性决定。因此，我们从这三个维度进行特征选择，在本文中一共选择了64个特征。

表2-1 代码特征

级别	名称	描述
类级别	属性成员变化	注释所在类中是否存在属性成员的变化?
	属性成员与注释的相关性	注释作用域内是否引用了变化的类的属性成员?
方法级别	方法声明变化	注释所在方法中是否存在方法声明的变化?如方法名, 方法返回类型, 方法参数列表等
	方法与注释的相关性	注释作用域内是否引用了变化的方法参数, 或者在方法返回类型变化的情况下, 包含返回语句?
语句级别	语句个数	注释作用域内包含的语句个数
	变化语句个数	注释作用域内包含的语句中变化的个数
	变化语句比例	注释作用域内变化语句个数与总的语句个数的比例
	语句变化类型	注释作用域内每种语句变化类型的个数, 本文关注的语句类型有12种, 分别为: <i>IF</i> 语句, <i>ELSE IF</i> 语句, <i>FOR</i> 语句, <i>Enhanced FOR</i> 语句, <i>WHILE</i> 语句, <i>CATCH</i> 语句, <i>TRY</i> 语句, <i>THROW</i> 语句, <i>PRINT</i> 语句, <i>LOG</i> 语句, 方法调用语句和变量声明语句。其中, 变化类型有: 新增, 删除和修改。共36种语句变化类型
	代码重构	代码变化是否为代码重构(包含9种同构类型)

2.5.1 代码特征

代码特征包括代码的上下文特征和代码变化的特征。我们分别从类级别和方法级别提取软件变化的上下文特征, 从语句级别提取软件变化的代码特征。代码特征列表如表2-1所示。

在基于面向对象的程序语言中, 类和方法作为软件变化的上下文环境存在。在包含软件变化的类中, 如果存在类的属性成员的变化, 并且在软件变化中引用了变化的类属性成员, 则软件变化中的代码行为将有可能发生变化。这时与代码关联的注释有可能需要进行相应的修改。类似地, 在包含软件变化的方法中, 如果存在方法的声明变化, 如方法名变更, 方法参数列表变化, 方法返回类型变化等, 并且在软件变化中引用了变化的方法参数, 或者包含了返回值语句, 则软件变化中的代码行为也有可能发生变化。这时与代码关联的注释也可能需要进行相应的修改。所以, 在我们的代码特征选择中, 选取了类和方法两种类型的软件变化上下文特征。

表 2-2 变化语句个数统计

变化语句个数	1-3	4-6	7-9	10-12	13-15	>15
注释变化	1891	866	536	382	282	762
注释未变化	23933	3709	1287	593	283	526
注释变化比例	7.3%	18.9%	29.4%	39.1%	49.9%	59.2%

表 2-3 语句变化类型统计

	注释	If	Else If	For	Enhanced- For	While	Catch	-
新增	变化	1752	212	147	121	172	596	-
	未变化	3404	379	219	332	206	991	-
	变化比例	34%	36%	40%	27%	46%	38%	-
修改	变化	868	180	71	40	109	238	-
	未变化	3321	550	203	173	177	740	-
	变化比例	21%	25%	26%	19%	38%	24%	-
删除	变化	1071	153	118	55	173	472	-
	未变化	1812	166	240	66	214	719	-
	变化比例	37%	48%	33%	45%	45%	40%	-
	注释	Try	Throw	Print	Log	Method Call	Variable Declaration	均值
新增	变化	432	360	72	840	3067	1690	-
	未变化	716	658	206	1994	8563	3663	-
	变化比例	38%	35%	26%	30%	26%	32%	33.72%
修改	变化	164	240	42	611	2299	1379	-
	未变化	389	903	279	3289	20254	8694	-
	变化比例	30%	21%	13%	16%	10%	14%	20.58%
删除	变化	330	290	43	615	1944	1192	-
	未变化	392	439	162	1123	5300	2514	-
	变化比例	46%	40%	21%	35%	27%	32%	36.50%

在代码特征中，另一类主要的特征为代码变化的特征。我们从语句的级别上提取代码变化特征。在软件变化中，语句变化的个数对代码和注释的一致性

具有显著的影响，通常的表现为：修改范围越大的代码段，其对应的注释需要同步修改的概率越大[?]。如表2-2所示，当语句变化的个数越多时，软件变化中的原注释越倾向于需要修改。因此，我们将软件变化的语句个数，语句变化个数以及语句变化占代码片段语句总数的比例加入到代码特征中。此外，不同类型的语句变化对代码和注释的一致性影响程度存在差异，为了量化不同类型的语句变化对代码和注释的一致性影响，我们对各种语句变化类型进行了统计，统计结果如表2-3所示。从表中可以看出，新增和删除While语句，删除Else If, Enhanced FOR和TRY语句，注释变化的比例都超过了45%。而新增或删除FOR语句，删除IF, CATCH和THROW语句，注释需要变化的比例也超过了40%。这些结果表明当软件变化的变化语句包含上述的变化类型时，注释需要修改的概率高于其他变化类型。另外，我们还发现当语句变化类型为新增或删除类型时，其注释的平均变化比例均高于修改类型。因此，我们将不同语句变化类型的个数以及它们所占比例加入到代码特征中。

表 2-4 代码重构类型

重构类型	描述
方法提取	将代码片段转换成一个方法，方法名称解释了该方法的作用
方法展开	将方法体展开到调用者中的代码块中，并移除该方法
方法重命名	改变方法的方法名
增加方法参数	在方法的参数列表中增加参数
减少方法参数	在方法的参数列表中移除参数
临时变量展开	使用简单的表达式代替临时变量，并将对临时变量的引用全都替换成表达式
字段封装	将公有字段修改为私有字段，并提供访问方法
断言引入	使用一段代码对程序状态进行设定，并使用断言判断程序是否符合预期
使用检测代替异常	当发生异常时，如果调用者可以使用条件语句检测进行避免，则使用条件语句代替异常捕获语句

当代码变化为代码重构时，代码变化的特征对代码和注释的一致性影响将会降低。其中，代码重构是指在不改变代码的功能和外部可见性的情况下，为

表2-5 注释特征

名称	描述
注释长度	对注释进行分词，并对分词列表进行计数
任务型注释	注释是否包含任务型标志(“TODO”, “FIXME”, “XXX”)
Bug 或者版本类型注释	注释是否包含“bug”或者版本类型的标志(“bug”, “fixed bug”, “version”)
软件变化所在类的注释和代码比例	软件变化所在类的注释行数占类的代码行数的比例
软件变化所在方法的注释和代码比例	软件变化所在方法的注释行数占方法的代码行数的比例
软件变化中注释和代码比例	软件变化中的注释行数占软件变化的代码行数的比例

了改善代码的结构，而对代码进行的修改。由于代码重构往往涉及范围较大的代码变化，而这些代码变化不会改变代码的行为。也就是说，虽然有很多语句发生了变化，如果在注释中只是对代码的功能进行描述，而没有关注具体的实现细节，这时注释往往不需要变化。另一种情形是在注释中提及了功能的具体实现细节，这时即使是代码重构构成的代码变化，如果实现细节变化了，注释也需要做相应的修改。因此，我们在进行特征选择时，选取了9种代码重构的特征(如表2-4所示)。

2.5.2 注释特征

注释特征如表2-5所示。其中，注释的长度影响注释对其关联的代码所实现功能的描述的有效性和准确性[?]。如果注释包含关于代码的实现细节的描述，那么，当相应的代码发生变化时，注释需要修改的概率也将增大。因此，我们将注释长度加入到注释特征中。另外，软件变化所处的注释环境通常也会对注释是否需要修改造成影响。比如，在一个注释充分的类或方法中，注释的描述信息对代码的覆盖面比较大，这时当注释关联的代码发生变化时，注释需要作出相应修改的概率也将增大。在注释环境特征选择中，我们考虑三个级别的注释密度，分别为类，方法和代码片段。

此外，在软件变化中，不同类型的注释是否需要修改受代码变化的影响是不同的。在注释中，一些特殊的关键词常用于标注不同目的的注释。这些标注信息对注释的类型分类具有重要作用[?]。我们将任务型注释关键字(“TODO”, “FIXME”, “XXX”)和Bug及版本类型关键字(“bug”, “fixed bug”, “version”)加入到注释特征中。当这些类型的注释所关联的代码发生变化时，注释更倾向于需要修改。比如，对于“TODO”类型的注释，开发人员在其关联的代码中增加了代码实现，则在新版本中很有可能已经完成了“TODO”类型的注释所需要的功能。这时开发人员应删除注释中的“TODO”标志，以表明这项代码编写任务已经完成。

2.5.3 代码和注释的关联特征

在现有的研究工作中，不同的研究者提出了很多关于如何编写质量良好的注释的建议[?, ?]。注释通常被认为需要包含代码实现的功能和目标描述，以及可选的代码实现背后的原理描述[?]。由此可知，注释的描述信息很可能会提及代码片段中包含的实体或者所调用的方法。此外，对于一个高质量的注释，注释的描述信息和代码之间应该具有较高的语义相似度。也就是说，注释和代码之间的相似度越高，注释和代码的关联性也将越高。

通常来说，注释和其关联的代码具有较高的相似度[?]。如果在代码变化过程中，注释和代码的相似度差异很明显，那么此时在新版本中，很可能出现代码和注释不相一致的情况，这也就意味着注释需要作出相应的修改。这里出现的一个问题是应如何度量代码和注释之间的语义相似度。目前计算文本之间语义相似度的方法主要有两种：一种是基于WordNet的语义相似度计算方法。WordNet是一个庞大的英语词汇数据库，不同词性的英语词汇被组织成同义词的网络。其中，同义词通过概念-语义和词汇关系相互联系^①。两个单词间的语义相似度则可以通过计算它们在WordNet构成的单词网络中的所在位置的距离衡量[?]。另一种是基于Word Embedding技术的语义相似度计算方法，其核心思想是通过嵌入一个线性的投影矩阵，将原始的One-Hot向量(除了一个词典索引的下标对应的方向上是1，其余方向上都是0)映射为一个稠密的连续向量，并通过一个语言模型的任务去学习向量的权重[?]。该方法学习得到的词向量具有这样的性质：语义上相似的单词其对应的词向量也相似。因此，两个单词的语义

^① <http://wordnet.princeton.edu>

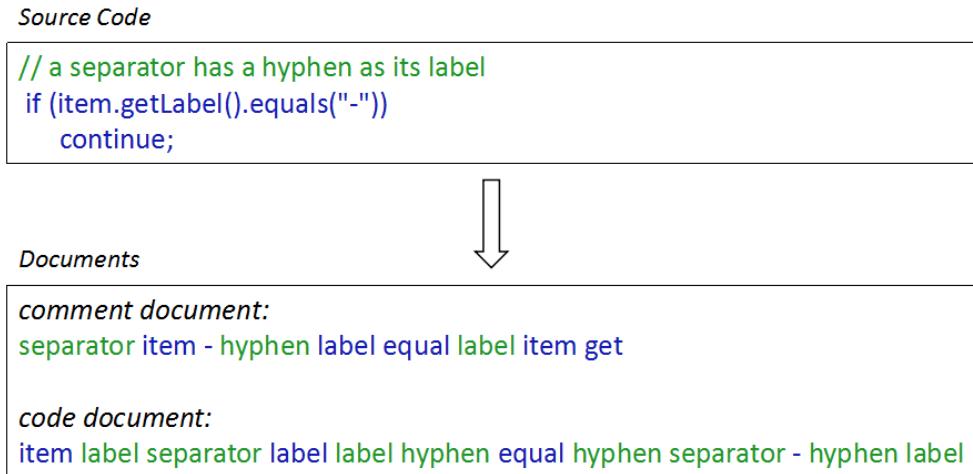


图 2-7 词向量模型的文档生成示例

相似度可以通过计算它们对应的词向量的距离得到。由于这两种方法都只是针对自然语言的文本的语义相似度计算方法，而在软件项目中，代码是通过编程语言编写的，注释则是采用自然语言的方式，两者之间存在语义的鸿沟。因此，我们需要一种可以有效衡量代码和注释之间的语义相似度的方法。在本文中选取了Ye等人提出的语义融合的词向量模型对代码和注释的相似度进行度量[?]。使用这种方式将有效消除代码和注释之间的语义鸿沟，具体步骤如下所示。

首先，我们对注释和代码的文本进行预处理，包括分词，去除停用词以及取词根等处理。其次，对注释和代码的文本信息进行融合。对于每一条注释中的单词，从其关联的代码中随机选取两个单词加入到该单词的后面，组合成新的注释文本信息。对于每一个代码片段中的单词，从其关联的注释中随机选取两个单词加入到该单词的后面，组合成新的代码文本信息。最后，我们将重新生成的注释和代码作为词向量模型的语料库。图2-7为语料库的文档生成示例(绿色字体为注释，蓝色字体为代码)。

在获得语料库之后，我们利用语料库训练词向量模型。基于Word Embedding技术的词向量模型主要分为两类，一类是Skip-Gram模型，另一类是CBOW模型。这两类模型的主要不同之处在于在进行词向量模型训练时，前者是通过给定的目标单词来预测其上下文的概率分布，而后者则是通过上下文信息来预测目标单词的概率分布[?, ?]。在本文中，采用的词向量模型为Skip-Gram模型，

模型的学习目标为最大化以下目标函数:

$$\sum_{i=1}^n \sum_{-k \leq j \leq k, j \neq 0} \log p(w_{i+j}|w_i) \quad (2.1)$$

其中, w_i 和 w_{i+j} 分别表示长度为 $2k + 1$ 的上下文窗口中的中心词和中心词的上下文(本文中取 $k = 2$, 即上下文窗口大小为5), n 代表语句的长度。式子 $\log p(w_{i+j}|w_i)$ 表示一个条件概率, 该条件概率由softmax函数定义, 如下所示:

$$\log p(w_{i+j}|w_i) = \frac{\exp(v_{w_{i+j}}'^T v_{w_i})}{\sum_{w \in W} \exp(v_w'^T v_{w_i})} \quad (2.2)$$

其中, v_w 表示输入向量, v'_w 表示模型中的单词 w 的输出向量。 W 表示所有单词的词汇。而 $p(w_{i+j}|w_i)$ 为在中心词 w_i 的上下文中出现的单词 w_{i+j} 的归一化概率。我们采用负抽样方法来计算这个概率[?]。

表 2-6 代码和注释的关联特征

名称	描述
旧代码和注释的相似度	变化前的代码和注释的语义相似度
新代码和注释的相似度	变化后的代码和注释的语义相似度
变化前后的代码和注释的相似度之差	变化前的代码和注释的语义相似度减去变化后的代码和注释的语义相似度
变化的语句变化前与注释的相似度	变化的语句在变化前与注释的语义相似度
变化的语句变化后与注释的相似度	变化的语句在变化后与注释的语义相似度
变化的语句在变化前后和注释的相似度之差	变化的语句在变化前和注释的语义相似度减去变化后和注释的语义相似度

通过以上方法得到词向量模型之后, 就可以得到每个单词的词向量表示。而为了计算代码和注释之间的语义相似度, 我们定义了以下三种相似度:

(1) 单词到单词: 给定两个单词 w_1 和 w_2 , 我们定义它们之间的语义相似度为这

两个单词对应的词向量的余弦相似度:

$$\text{sim}(w_1, w_2) = \cos(\mathbf{w}_1, \mathbf{w}_2) = \frac{\mathbf{w}_1^T \mathbf{w}_2}{\|\mathbf{w}_1\| \|\mathbf{w}_2\|} \quad (2.3)$$

(2) 单词到语句: 给定一个单词 w 和一条语句 S , 它们之间的语义相似度为单词 w 和语句 S 中的单词 w' 相似度最大的单词之间的语义相似度:

$$\text{sim}(w, S) = \max_{w' \in S} \text{sim}(w, w') \quad (2.4)$$

(3) 语句到语句: 两条语句 S_1 和 S_2 的语义相似度定义如下所示:

$$\text{sim}(S_1, S_2) = \frac{\text{sim}(S_1 \rightarrow S_2) + \text{sim}(S_2 \rightarrow S_1)}{2} \quad (2.5)$$

其中,

$$\text{sim}(S_1 \rightarrow S_2) = \frac{\sum_{w \in S_1} \text{sim}(w, S_2)}{n} \quad (2.6)$$

n 为 S_1 中的单词个数。

在代码和注释的关联特征选择过程中, 我们分别计算变化前的代码和注释的语义相似度、变化后的代码和注释的语义相似度、变化的语句在变化前后和注释的语义相似度、代码在变化前后和注释的相似度之差、以及变化的语句在变化前后和注释的相似度之差。将这六个相似度度量作为代码和注释的关联特征。如果变化前后的代码与注释的相似度相差较大, 则在新版本中, 注释也很有可能需要更新。同理, 如果变化的语句在变化前后和注释的相似度有很大差异, 注释所描述的语句将有很大概率已经不存在或者修改过, 则这个时候注释也往往需要进行更新。表2-6中展示了所有的代码和注释的关联特征。

2.6 一致性检测实验设置与结果评估

在本节中将介绍一致性检测的实验设计, 首先介绍实验数据的选取过程, 然后介绍一致性检测模型的训练过程, 最后给出检测模型的性能评估标准以及实验结果。

2.6.1 数据收集

表2-7 实验数据：5个开源项目

项目名	时间跨度	提交个数	软件变化个数	注释	
				变化	未变化
JEdit	2000/01-2016/03	2133	4750	662	4088
OpenNMS	2002/05-2009/11	1558	5585	941	4644
JAMWiki	2006/06-2013/03	1262	3756	467	3289
EJBCA	2001/11-2017/01	4705	19800	2442	17358
JHotDraw	2000/10-2016/01	251	1159	83	1076
Total	-	9909	35050	4595	30455

我们从开源项目中选择了5个项目作为我们实验的数据，具体如表2-7所示。这5个开源项目在Sourceforge^①中的评分均超过了4.8分(满分5分)，并且它们在软件工程领域经常被选作实验数据。此外，这5个开源项目分别代表了不同的工程领域，包括文本编辑器(JEdit^②)，管理系统(OpenNMS^③)，维基百科搜索引擎(JAMWiki^①)，企业安全认证系统(EJBCA^②)，以及二维图像处理软件(JHotDraw^③)。

如表2-7所示，在这5个项目中，我们过滤掉注释质量低的软件变化，包括(1)注释单词个数小于3个；(2)注释内容为代码；(3)注释内容为无意义的符号。在过滤之后，一共收集了9909个提交共35050个软件变化。在这些软件变化中，注释随着代码一起变化的个数为4595个，注释未变化的个数为30455个。我们将这些软件变化分为两部分：(1)从每个项目中随机提取400个软件变化，共2000个

① Sourceforge,https://sourceforge.net/,2018

② JEdit,http://www.jedit.org/,2018

③ OpenNMS,http://www.opennms.org/,2018

① JAMWiki,http://www.jamwiki.org/,2018

② EJBCA,http://www.ejbcna.org/, 2018

③ JHotDraw,http://www.jhotdraw.org/,2018

表2-8 实验中使用的数据集

数据集	软件变化	注释	
		变化	未变化
<i>Dataset₁</i>	33050	4320	28730
<i>Dataset₂</i>	2000	275	1725

软件变化作为一个数据集，标记为*Dataset₁*，用于方法的研究问题2的验证。

(2)所有不在*Dataset₁*中的数据作为另一个数据集，标记为*Dataset₂*，用于模型的训练和验证(如表2-8所示)。

2.6.2 模型训练

我们将代码和注释的一致性检测问题当作一个二分类问题。在一个软件变化中，如果随着代码变化注释需要作出修改，则认为此时代码和注释是不一致的，即如果保留原有注释且不作任何修改的情况下，注释的描述与修改后的代码的实现功能不相符合。我们将这部分数据当作数据集的正样本。如果在代码修改的情况下，注释不需要作出任何修改，则认为此时代码和注释是一致的。我们将这部分数据作为数据集的负样本。我们的目标就是从这些数据集中学习得到一个分类模型，以区分正负样本。

在传统的机器学习领域中，建立分类模型的算法主要有决策树算法，贝叶斯分类算法，支持向量机算法，增强学习算法以及集成学习算法等。其中，决策树算法和支持向量机受噪声影响明显[?, ?]，而我们的数据集来源于开源项目，且时间跨度比较大(大于10年)，不可避免地会包含一些噪声。所以这两种类型的算法不适用于我们的模型学习。而贝叶斯分类算法的分类准确率比较低，所以在本文中也没有采用该方法。增强学习算法和集成学习算法对我们的模型学习有较好的效果，其中又以集成学习算法表现更佳。在集成学习算法中，使用比较普遍的为随机森林算法。随机森林算法模型是一种集成学习模型，采用决策树作为其基分类器，每个决策树由独立随机采样的样本和特征决定。在分类阶段由所有决策树共同投票决定最终结果，且随机森林具有很好的抗噪声能力以及不容易出现过拟合问题[?]。所以，在实验中，我们选择随机森林算法进行模型训练。

在随机森林算法中，可调节的参数主要有决策树数量，特征采样个数以及样本采样比例。我们通过参数迭代的方式选取出一组最优的参数。其中，决策树数量选择为300，特征采样个数为8，样本采样比例为30%。

2.6.3 模型评估标准

本文的主要目标是在一次代码提交中，根据修改的代码自动判断原有的注释是否需要修改。因此，我们在对模型的评估中主要关注两个方面的问题：第一，一致性检测模型的准确率能达到多少？第二，代码和注释的一致性检测在实际应用中是否对开发和维护人员有所帮助。

1: 一致性检测模型的准确率能达到多少？

为了验证模型的准确率，我们选择 $Dataset_1$ 作为模型训练和验证的数据集，并采用十折交叉验证的方法对模型进行评估。在对代码和注释的一致性判定模型的准确率评估中，我们主要考察6个度量指标：正例精确度($precision_1$)，正例召回率($recall_1$)和正例F1值($F1 - Score_1$)以及反例精确度($precision_0$)，反例召回率($recall_0$)和反例F1值($F1 - Score_0$)。其中，精确度表示当分类器将一个实例分类为某类时，有多大概率可以认为这次分类是准确的。召回率表示分类器对某种类别的实例集合可以正确分类的比例。F1值同时考虑精确度和召回率，通常是对精确度和召回率的一种折中的度量，常作为评价一个模型的综合性能被许多软件工程领域的论文中所采用。这6个度量指标定义如下：

$$precision_1 = \frac{ActualPositiveIns \cap EstimatedPositiveIns}{EstimatedPositiveIns} \times 100\% \quad (2.7)$$

$$precision_0 = \frac{ActualNegativeIns \cap EstimatedNegativeIns}{EstimatedNegativeIns} \times 100\% \quad (2.8)$$

$$recall_1 = \frac{ActualPositiveIns \cap EstimatedPositiveIns}{ActualPositiveIns} \times 100\% \quad (2.9)$$

$$recall_0 = \frac{ActualNegativeIns \cap EstimatedNegativeIns}{ActualNegativeIns} \times 100\% \quad (2.10)$$

$$F1 - Score_1 = \frac{precision_1 \cdot recall_1}{precision_1 + recall_1} \quad (2.11)$$

$$F1 - Score_0 = \frac{precision_0 \cdot recall_0}{precision_0 + recall_0} \quad (2.12)$$

其中, $ActualPositiveIns$ 表示正例的个数, $ActualNegativeIns$ 表示反例的个数, $EstimatedPositiveIns$ 表示分类器分类为正例的个数, $EstimatedNegativeIns$ 表示分类器分类为反例的个数。

2: 代码和注释的一致性检测在实际应用中是否对开发和维护人员有所帮助?

为了验证我们的模型是否对开发和维护人员在现有项目中寻找不一致的注释是否有帮助, 我们选择了数据集 $Dataset_2$ 作为我们的验证集, 通过模型对其进行分类, 并邀请9位志愿者对模型识别为不一致的注释进行验证。

2.6.4 一致性检测模型实验结果

1: 一致性检测模型的准确率能达到多少?

表 2-9 一致性模型评估结果

特征	$precision_1$	$recall_1$	$F1-Score_1$	$precision_0$	$recall_0$	$F1-Score_0$
代码特征	83.6%	58.8%	0.691	93.9%	98.2%	0.960
注释特征	47.3%	12.6%	0.199	87.8%	97.8%	0.925
代码和注释 关联特征	66.5%	38.1%	0.381	91.0%	97.0%	0.939
代码特征+ 关联特征	86.8%	60.6%	0.713	94.1%	98.6%	0.956
所有特征	88.4%	61.5%	0.726	94.3%	98.7%	0.965

本节按照上一节中提出的6个模型评估度量给出代码和注释一致性检测模型的评估结果。首先, 我们分别选择代码特征, 注释特征以及代码和注释的关联特征作为模型的特征, 观察不同类型的特征对模型评估结果的影响。然后, 我们将不同类型的特征相结合, 观察在考虑了多种维度的特征后, 模型的检测效果是否有所提高。具体结果如表2-9所示。从表中可知, 在代码特征, 注释特征以及代码和注释的关联特征中, 代码特征在代码和注释的一致性检测中效果最好, 代码和注释的关联特征次之, 注释特征相对来说起到的作用较小。而从各种类型的特征的结合结果来看, 同时考虑三种维度的特征, 一致性检测效果最

好，这说明了在特征选择中，同时考虑三种维度的特征，可以对代码和注释的一致性作出最好的检测。

从表2-9中，我们发现正例的召回率比较低(61.5%)，这主要是由于模型的训练样本不平衡导致的。如表2-8所示，正负样本比例约为1:6.6。由此可见，我们的二类别分类问题为不平衡数据集的分类问题。一般解决不平衡分类问题的方式有三种：第一种为对少数类的样本进行过采样，即在样本采样环节提高少数类样本的采样概率，来达到平衡数据集中正负样本的比例。这种采样方式由于是简单的对少数类的同一个样本进行多次提取，在模型训练中容易出现过拟合的现象；第二种是对多数类的样本进行欠采样，即在样本采样环节降低多数类样本的采样概率，以平衡数据集中正负样本的比例。这种采样方式缩小了进行模型训练的样本数量，不能充分利用数据集进行建模；最后一种为代价敏感方法，其通过调整少数类和多数类的误分类代价，即在模型构建过程中，增大少数类的误分类代价，使分类模型偏向于选择少数类的分类结果。由于这种方法没有直接对数据集进行操作，而是通过对不同类别的样本赋值不同的权重以达到数据集的平衡，所以可以很好地克服前两种方法的缺陷[?]。因此，在本文中选择最后一种方法作为我们的不平衡分类问题的解决方案。具体地，我们通过将代价矩阵引入模型训练中，并修改随机森林的决策树中不纯度度量和分类判别公式，将我们的分类模型修改为代价敏感的分类模型。其中，代价矩阵在形式上如下所示：

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix}$$

其中， C_{ij} 表示分类器把类别为*i*的样本误分类为*j*的代价。一般来说， $C_{00} = C_{11} = 0$ 。我们修改随机森林的决策树的不纯度度量，由于在本实验中使用的是*Gini*不纯度，具体到我们的二分类问题，修改后的不纯度度量公式为：

$$Gini(D) = 1 - \left(\frac{|c_0| \times C_{01}}{|D'|} \right)^2 - \left(\frac{|c_1| \times C_{10}}{|D'|} \right)^2 \quad (2.13)$$

其中，

$$|D'| = |c_0| \times C_{01} + |c_1| \times C_{10} \quad (2.14)$$

其中, D 为样本的集合, $|D|$ 为集合中样本的个数, $|c_0|$ 为集合中负样本的个数, $|c_1|$ 为集合中正样本的个数。

决策树中叶子结点的分类标签计算公式修改为:

$$label = \underset{i}{\operatorname{argmax}} (|c_i| \times C_{ij}) \quad (2.15)$$

其中,

$$j = |1 - i| \quad (2.16)$$

在实验中, 我们将代价矩阵中的 C_{00} 和 C_{11} 设为0, 并通过调整 C_{01} 和 C_{10} 的值, 来观察正例的精确度, 召回率和F1值的变化情况。具体如图2-8所示。其中, 图2-8的横轴表示 $C_{01} : C_{10}$ 的比值, 纵轴表示准确率, 蓝色折线表示正例的精确度, 红色折线表示正例的召回率, 绿色折线表示正例的F1值。

从图2-8中可知, 当 $C_{01} : C_{10} = 1 : 6$ 时, 正例的F1值达到最大值(0.759), 这时正例的精确度为77.2%, 召回率为74.6%。模型评估的最终结果如表2-10所示, 通过对比表2-9中所有特征的评估结果和表2-10中的评估结果可知, 虽然在正例的精确度上从88.4%降到了77.2%, 但在正例的召回率上有明显的提

图 2-8 不同代价矩阵下的模型评估结果

表 2-10 一致性模型评估结果（引入代价矩阵后）

$precision_1$	$recall_1$	$F1 - Score_1$	$precision_0$	$recall_0$	$F1 - Score_0$
77.2%	74.6%	0.759	96.42%	97.22%	0.968

升(从61.5%提高到74.6%)，并且正例的F1值也从0.726上升到0.759。而在反例的评估结果上则没有明显差别。这说明通过引入代价矩阵，可有效调节模型中训练集的不平衡性。

2: 代码和注释的一致性检测在实际应用中是否对开发和维护人员有所帮助?

对于第二个研究问题，我们使用数据集 $Dataset_1$ 进行模型训练，并将数据集 $Dataset_2$ 作为测试集，使用训练好的模型对这部分数据的注释进行一致性检测。在数据集 $Dataset_2$ 中，包含275个变化的注释以及1725个未变化的注释。通过模型的分类，有279个注释检测为与代码不一致，其中有210个注释在原项目中已作了修改，另外69个注释在原项目中未作修改，我们邀请9个志愿者对这69个注释进行验证。首先，我们将这9个志愿者分为3组，每组对这69个注释进行一轮验证，即每个志愿者验证23个注释。对于每一个注释，我们可以收集到3个答案。在对每个注释进行验证时，我们采用在线验证的方式，验证页面如

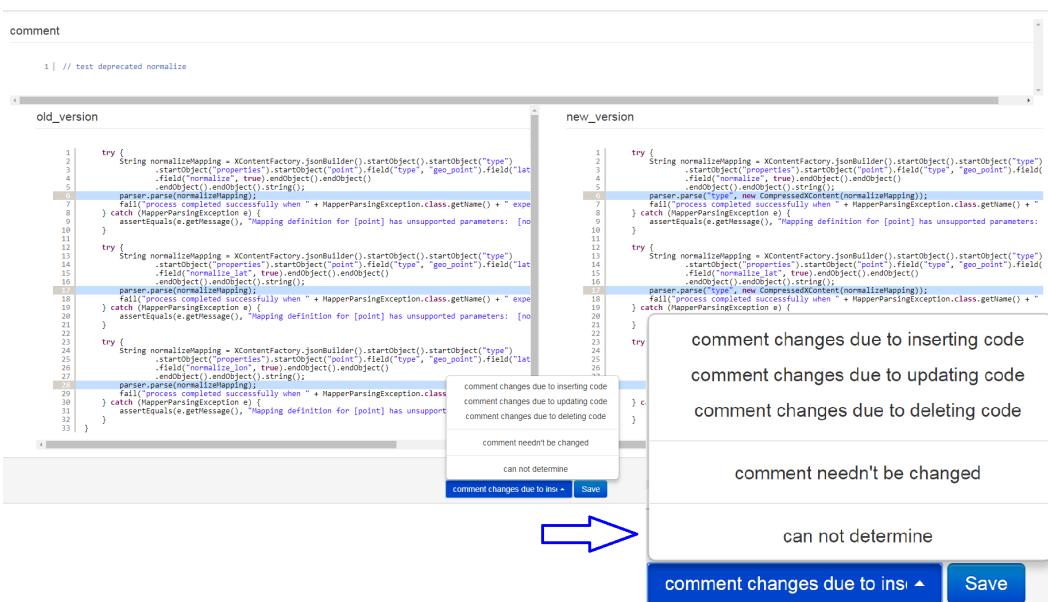


图 2-9 数据验证界面

表 2-11 验证结果类型

编号	描述
O_1	新增代码引起的注释与代码的不一致
O_2	修改代码引起的注释与代码的不一致
O_3	删除代码引起的注释与代码的不一致
O_4	注释与代码保持一致
O_5	无法确定注释与代码是否一致

表 2-12 验证结果

Group	O_1	O_2	O_3	O_4	O_5
Group 1	15	7	7	36	4
Group 2	17	6	5	33	8
Group 3	20	7	4	32	3
最终结果	19	6	6	31	7

图2-9所示，验证页面展示软件变化中变化前的注释，变化前的代码以及变化后的代码等信息。在验证选项中，有5个选项供参与者选择，具体说明如表2-11所示。最后，对于每一个注释是否与代码一致的验证，采取少数服从多数的原则，验证结果如表2-12所示。从结果中可以看出，我们的模型最后检测正确的与代码不一致的注释的个数为241 (210+19+6+6)个。在这241个不一致的注释中，有10%的注释在原项目中未作修改。由此可以看出，我们的模型可以帮助开发和维护人员在现有项目中检测出不一致的注释。

2.6.5 一致性检测影响因素讨论

在实验中，我们的训练集是直接从开源项目中提取的，且由于这部分数据数量较多，我们没有人力进行逐条验证。在训练集中不可避免地包含一部分噪声，这些噪声可能对模型的准确率造成影响。而在进行数据提取时，我们采用启发式规则检测注释作用域。在一些情况下，启发式规则倾向于扩大注释的作用域。在这种情况下，有可能将一些与注释不想关的代码加入到软件变化中，影响模型的准确率。另外，由于我们在实验中选择的项目的编程语言均为Java，

没有验证在其他编程语言项目中模型的性能。

2.7 本章小结

在本章中，我们提出了一种数据驱动的代码和注释一致性检测方法。该方法采用随机森林算法进行模型构建，并从代码，注释以及代码和注释的关系等多个维度进行特征提取。实验中，我们采集了5个开源项目的代码提交数据进行模型训练和验证。实验结果显示我们的模型在正样本中精确度达到77.2%，召回率达到74.6%，F1值为0.759。此外，我们从2000个注释中找到了241个与代码不一致的注释，并有31个注释在原项目中未作出修改。这表明我们的模型可以有效地帮助开发和维护人员在现有项目中发现与代码不一致的注释，辅助开发和维护人员评估代码提交的质量。

第3章 基于关键类判定的代码提交理解辅助方法

软件代码的提交注释是开发人员理解代码提交的主要文档。开发人员通过阅读提交注释，来对代码提交进行快速理解。我们在对提交注释进行分析时发现，在一部分提交注释中，常会出现一个或几个在代码提交中涉及修改的类的类名。在我们的进一步研究中发现，在提交注释中被提及的类通常为被核心修改的类，本文中称其为关键类。关键类可有效引导开发者聚焦于在代码提交中需重点关注的修改，快速理解代码提交中的代码修改，从而提高代码提交理解的效率。在本文中，提出一种自动识别关键类的方法-ICC(Identifying Core-Class)，辅助开发和维护人员理解代码提交。该方法从代码耦合，代码修改和提交类型三个维度共提取出21种特征，以及结合自然语言处理的技术，对类的代码修改进行向量化，加入到特征集中，用以关键类的判定。在方法中，我们从版本控制库中获取大量的代码提交数据，并将在提交注释中提及的类标记为关键类，未提及的类标记为非关键类，利用这些样本训练关键类判定模型。最后，将训练好的判定模型用于代码提交的关键类判定。

3.1 关键类判定中的基本概念

本节中，分别给出了提交，类，关键类和非关键类的定义，具体定义如下所示：

定义1(提交(Commit, Cm)): 定义为四元组， $Cm=(m, S, K, N)$ 。其中， m 为提交注释； S 为一次代码提交中涉及修改的类的集合； K 表示一次代码提交中核心修改的类的集合； N 表示一次代码提交中非核心修改的类的集合。

定义2(类(Class, C)): 定义为六元组， $C=(k, i, o, l, m, mc)$ ，其中， k 为关键类标识，指示当前类是否为关键类； i 表示类入度，指该类在本次提交中引用其他类的个数； o 为类出度，指该类在本次提交中引用它的类的个数； l 表示修改代码行数，为类中涉及修改的代码行数； m 为方法数，指示类中方法的数量； mc 表示修改方法数，为类中涉及修改的方法的数量。

定义3(关键类与非关键类集合(K, N)): 关键类(K)是提交中被核心修改的类；

非关键类(N)是为了完成关键类的修改而进行依赖性改动的类。在代码提交中，有可能存在一个或多个关键类。

3.2 关键类判定模型的特征提取

本节详细介绍了关键类判定模型的特征提取。我们从三个维度一共提取了21种特征，这些特征为：代码耦合特征，代码修改特征以及提交类型特征。代码耦合特征描述在一次提交修改中多个类之间的耦合关系，代码修改特征用以度量类中的代码修改的规模，提交类型特征将提交按照不同目的进行分类，用以刻画在不同类型的提交中，各种修改类型的类成为关键类的概率。在特征描述中所使用到的符号如表3-1所示。

3.2.1 代码耦合特征

代码耦合特征是指在一次代码提交中，多个类的修改之间的代码耦合关系。由于代码修改具有传递性，当两个类之间存在耦合关系时，一个类的修改有可能会引起另一个类的修改[?]。在一次代码提交中，非关键类的修改往往是由关键类的修改引起的。即关键类的代码修改向非关键类中传递，且这种修改传递具有方向性，其在类之间的代码结构关系中表现为代码耦合关系。因此，我们可以从多个类中的代码耦合关系出发，提取代码耦合特征进行关键类判定。

在代码耦合特征中，我们使用类的入度和出度来衡量一个类与其他类之间的耦合关系。一个类的入度是指提交中引用该类的其他类的个数，出度是指在该类中引用提交中的其他类的个数。同时，我们还考虑了提交中不同类之间的出度和入度的关系。假设当前类的入度为提交中所有类的入度的最大值，意味着当前类经常被其他类引用，那么这个类的修改很大概率会引起其他类的修改，其很有可能作为此次提交的关键类存在。同理，如果一个类的出度很大，也就意味着这个类受其他类的修改的影响比较大，这个类有比较高的概率成为非关键类。一个类的入度和出度的不同组合也有可能影响关键类的判别。如一个类的入度和出度都很大，说明这个类影响其他类的修改以及受其他类的修改的影响都比较大，这时没有明显倾向说明一个类是否是关键类。如果一个类的入度很大，而出度为0，这个类有很大概率成为关键类；相反地，如果一个类的出度很大，而入度为0，这个类更倾向于非关键类。

表 3-1 符号说明

符号	形式化表示	描述
I_{curr}	$C.i$	当前类的入度;
I_{max}	$\max\{C_j.i\}, 1 \leq j \leq S $	提交中所有类中拥有的最大入度;
I_{ave}	$\sum_{j=1}^{ S } C_j.i / S $	提交中所有类的平均入度;
I_{zero}	$\text{boolean}(I_{curr} == 0)$	类的入度是否为0;
O_{curr}	$C.o$	当前类的出度;
O_{max}	$\max\{C_j.o\}, 1 \leq j \leq S $	提交中所有类中拥有的最大出度;
O_{ave}	$\sum_{j=1}^{ S } C_j.o / S $	提交中所有类的平均出度;
O_{zero}	$\text{boolean}(O_{curr} == 0)$	类的出度是否为0;
S_{curr}	$C.l$	类修改的语句条数;
S_{max}	$\max\{C_j.l\}, 1 \leq j \leq S $	提交中所有类中修改最多的语句条数;
S_{ave}	$\sum_{j=1}^{ S } C_j.l / S $	提交中所有类平均修改的语句条数;
$M_{methincla}$	$C.m$	类中方法的总数;
$M_{changedmethincla}$	$C.mc$	类中涉及修改的方法的数量;
$M_{methincomm}$	$\sum_{j=1}^{ S } C_j.m$	提交中方法的总数;
$M_{changedmethincomm}$	$\sum_{j=1}^{ S } C_j.mc$	提交中涉及修改的方法的总数;
CC_{type}	$CC_{type} = \begin{cases} new \\ changed \\ remove \end{cases}$	类的修改类型,取值为新增,修改,删除;
$C_{forwardengineering}$	--	提交的类型是否为前向工程;
$C_{reengineering}$	--	提交的类型是否为逆向工程;
$C_{correctiveengineering}$	--	提交的类型是否为纠错工程;
$C_{management}$	--	提交的类型是否为非代码修改类型;

表3-2展示了所有的代码耦合特征。其中, CF_1 表示当前类的入度; CF_4 表示当前类的出度; CF_2 和 CF_3 分别为当前类的入度与提交中类的最大入度的比值以及提交中类的平均入度的比值; CF_5 和 CF_6 为当前类的出度与提交中类的最大出度的比值以及提交中类的平均出度的比值; CF_7 表示当前类的入度是否为0; CF_8 表示当前类的出度是否为0。

3.2.2 代码修改特征

在一次提交中, 类的代码修改量对于关键类的判别具有很强的指向作用。

表 3-2 代码耦合特征

类型	特征	描述
代码耦合特征	CF_1	I_{curr}
	CF_2	I_{curr}/I_{max}
	CF_3	I_{curr}/I_{ave}
	CF_4	O_{curr}
	CF_5	O_{curr}/O_{max}
	CF_6	O_{curr}/O_{ave}
	CF_7	$if\ I_{zero} = true, CF_7 = 1; otherwise, CF_7 = 0$
	CF_8	$if\ O_{zero} = true, CF_8 = 1; otherwise, CF_8 = 0$

关键类在一次提交中往往是被主要修改的类，非关键类中的代码修改通常是为了与关键类保持一致，而作出的辅助性修改。这种关系体现在代码修改量上则为关键类修改的代码量通常要比非关键类多。类的修改代码量可以使用类中方法的修改个数和代码的修改行数来度量。同时，我们还在代码修改特征中引入了代码相对修改量的概念，即在一次提交中，类中代码修改量与提交中涉及的类的平均代码修改量的比值。代码修改特征如表3-3所示。

我们从代码修改维度选择了8个特征，如表3-3中的特征 MF_1-MF_8 所示。其中， MF_1 表示当前类的修改代码行数，特征的值使用区间化表示。如当前类修改的代码行数为10行，则特征 MF_1 的值为3。 MF_2 表示当前类的代码修改行数与提交中类的最大代码修改行数的比值，该值也是当前类中的代码修改行数的归一化结果。 MF_3 表示当前类的代码修改行数与提交中类的平均修改行数的比值， MF_3 用于衡量当前类的代码修改量的程度。 MF_4 为当前类的修改方法的个数，与 MF_1 相似，该特征的值也使用区间化表示。 MF_5 用于衡量类中修改的方法占类中方法的比例。 MF_6 和 MF_7 分别为当前类的方法修改数与提交中类的平均方法修改数的比值以及类中平均方法数的比值，这两个特征用于衡量当前类的修改方法数在整个提交中的水平。最后， MF_8 代表当前类的修改类型，类的修改类型分为新增，修改和删除三种类型。

3.2.3 提交类型特征

在一次提交中，代码修改所要达到的目标有几种类型：(1)添加新功能(前向

表 3-3 代码修改特征

类型	特征	描述
代码修改特征	MF_1	$1 \leq S_{curr} \leq 5, MF_1 = 1; 6 \leq S_{curr} \leq 9, MF_1 = 2; 10 \leq S_{curr} \leq 19, MF_1 = 3; 20 \leq S_{curr} \leq 29, MF_1 = 4; 30 \leq S_{curr}, MF_1 = 5$
	MF_2	S_{curr}/S_{max}
	MF_3	S_{curr}/S_{ave}
	MF_4	$1 \leq M_{changedmethincla} \leq 5, MF_4 = 1; 6 \leq M_{changedmethincla} \leq 9, MF_4 = 2; 10 \leq M_{changedmethincla} \leq 19, MF_4 = 3; 20 \leq M_{changedmethincla} \leq 29, MF_4 = 4; 30 \leq M_{changedmethincla}, MF_4 = 5$
	MF_5	$M_{changedmethincla}/M_{methincla}$
	MF_6	$M_{changedmethincla}/M_{changedmethincomm}$
	MF_7	$M_{changedmethincla}/M_{methincomm}$
	MF_8	$if CC_{type} = new, MF_8 = 0; else if CC_{type} = change, MF_8 = 1; else if CC_{type} = remove, MF_8 = 2$

工程); (2)删除过时代码(逆向工程); (3)修复代码缺陷(纠错工程); (4)非可执行文件修改(非代码修改类型), 如配置文件, 说明文件等。根据提交的类型不同, 提交中被核心修改的类也会有所不同。如: 提交类型为添加新功能, 关键类通常为新增的类。因此, 如果已知一个提交的类型, 可以将当前提交的关键类限定在一个更小的范围内, 然后利用代码耦合特征和代码修改特征进一步判定一个类是否为关键类。

表 3-4 提交分类及关键字列表

提交分类	关键字
前向工程	implement,add,request,new,test,start,include,initial...
逆向工程	optimiz,adjust,update,delet,remov,chang,refactor...
纠错工程	bug,issue,error,correct,proper,deprecate,broke...
非代码修改类型	clean,license,merge,release,structure,style,copyright...

本文参考了已有研究[?]对提交进行分类的方法，该方法通过识别提交注释中的关键字对提交类型进行分类。比如，当提交注释中包含“new”，“add”，“create”等关键字时，则将该提交分类为添加新功能(前向工程)类别。本文采用相同的方法对提交进行分类，我们在表3-4中给出了提交类型及对应的关键字列表。提交类型特征如表3-5所示，其中， TF_1 - TF_4 四个特征分别对应提交的四个类别。 CI_1 为提交中涉及的类的个数。

3.2.4 词向量特征

在自然语言处理领域中，常使用词向量表示单词的语义信息。词向量模型主要有两种：一种是Skip-Gram模型，另一种是CBOW模型。这两类模型的主要不同之处在于在进行词向量模型训练时，前者是通过给定的目标单词来预测其上下文的概率分布，而后者则是通过上下文信息来预测目标单词的概率分布[?, ?]。在本文中，采用的词向量模型为Skip-Gram模型作为我们的词向量模型。

首先，我们选择代码提交中的源代码作为模型语料库。仿照自然语言处理的方法，将一个类看作是一篇文档，类中的字符串看作是由单词组成的序列。在对字符串进行分词时，有以下几个步骤：(1)根据分隔符将字符串切分成一个单词集合；(2)按照驼峰命名规则，将变量名切分成多个独立的单词；(3)对每个单词的字符，转化成小写字母，并提取单词词根。然后，我们根据Skip-Gram模型的训练方法训练词向量模型。我们设置模型的上下文窗口大小为5，词向量长度为100。得到词向量模型后，实际上，我们得到了语料库中每个单词的词向量表示。最后，我们从代码提交中提取每个类变化的语句，并对它们进行分词，

表 3-5 提交类型特征

类型	特征	描述
提交类型特征	TF_1	$if C_{forwardengineering} = true, TF_1 = 1; otherwise, TF_1 = 0$
	TF_2	$if C_{reengineering} = true, TF_2 = 1; otherwise, TF_2 = 0$
	TF_3	$if C_{correctiveengineering} = true, TF_3 = 1; otherwise, TF_3 = 0$
	TF_4	$if C_{management} = true, TF_4 = 1; otherwise, TF_4 = 0$
	CI_1	$1 \leq S \leq 5, CI_1 = 1; 6 \leq S \leq 9, CI_1 = 2; 10 \leq S \leq 19, CI_1 = 3; 20 \leq S \leq 29, CI_1 = 4; 30 \leq S , CI_1 = 5$

得到每个类变化的单词集合。通过词向量模型，获得每个单词的词向量，并将它们进行累加操作，得到每个类的词向量表示。最终，将词向量加入到特征集中，词向量的每一维作为类的一个特征。

3.3 机器学习算法选择及样本优化处理

本文提出的关键类识别方法*ICC*是基于有监督的机器学习算法。具体地，我们将提交中一个类是否为关键类视为二分类问题，类别“0”代表非关键类，类别“1”代表关键类。在开源项目的提交中，我们提取代码耦合特征，代码修改特征以及提交类型特征用以关键类的识别。然后选择一种分类模型根据提取的特征进行训练。最后根据训练好的模型对提交中的类进行关键类判定。

3.3.1 机器学习算法选择

机器学习的分类模型中，包含多种类型的分类算法，包括决策树，贝叶斯网络，支持向量机，随机森林等。我们通过比较各种算法在数据集中的分类表现，选择分类效果最好的随机森林作为我们的分类模型。随机森林利用bootstrap重抽样方法从原始样本中抽取多个样本，对每个样本进行决策树建模，然后组合多棵决策树的预测，通过投票得出最终预测结果[?]。它在分类中有比其他分类器更好的分类效果，且对噪声具有良好的鲁棒性。由于我们的数据集来源于众多的开源项目，不可避免地存在一些噪声。因此，随机森林良好的鲁棒性是我们选择它作为分类器建模的一个重要因素。另外，由于随机森林最终的决策综合了多个独立的分类器的预测结果，这使得随机森林不容易出现过拟合，在对未知数据的预测中具有良好的泛化能力。且随机森林只需调整少数几个参数，就可以得到一个表现良好的分类模型，这降低了模型调参的复杂度。综上所述，我们选择使用随机森林对我们的分类器进行建模。

3.3.2 样本优化处理

本文中用于模型训练及验证的数据来源于开源项目的代码提交数据。而由于项目间的代码提交数据质量参差不齐，在进行模型训练之前，我们需要对数据进行预处理。我们在对大量的代码提交进行观察，设置了以下几条过滤规则：(1)过滤掉提交注释中没有提到关键类信息的代码提交；(2)过滤掉在提交注

释中提到的类数量等于代码提交中涉及修改的类的总数的代码提交; (3)过滤掉只涉及一个类修改的代码提交; (4)过滤掉超过二十个类修改的代码提交, 此类提交通常是由多个普通的提交组合而成, 一般出现在版本更新的代码提交中; (5)过滤掉注释信息过短的提交(小于3个单词), 这类提交注释由于缺乏有效信息, 难以判断出现在注释中的类是否为关键类; (6)过滤掉注释信息过长的提交(大于200个单词), 这类提交中往往提到多个类, 并罗列了提交中大部分琐碎的修改, 无法判断在注释中出现的类为关键类。

在机器学习算法的应用中, “坏样本”对于模型的性能影响很大。其中, “坏样本”是指在样本集中存在两个特征向量完全相同的样本, 但它们的类标签却不同。其有可能是因为数据集的特征细分的粒度不够或者是数据集本身包含错误数据等原因造成的。在我们的实验中发现, “坏样本”对模型的分类准确率造成很大的影响, 因此, 必须剔除掉样本集中的“坏样本”。“坏样本”的过滤算法如算法2所示。

在算法2中, 将负样本 N , 正样本 P , 总样本 T 以及总的提交集合 C 作为输入, 输出为过滤后的样本集合 $new_SampleSet$ 以及提交集合 $new_CommitSet$ 。对于每一个正样本 ps 以及每一个负样本 ns (第5, 6行), 计算两者的余弦相似度 $cosineSimilar$ (第7行), 如果相似度为1, 则将这一对样本加入到坏样本集合 $badSample_Set$ 中(第8, 9行)。遍历完所有的正样本和负样本后, 如果坏样本集合不为空(第13行), 则对于每一个提交 $Commit$, 如果提交中涉及修改的类的集合为坏样本集合的子集, 则在提交集合中移除当前提交(第14-16行)。然后, 在样本集合中移除坏样本(第19, 20行)。最后, 将过滤之后的样本集合 $new_SampleSet$ 以及提交集合 $new_CommitSet$ 作为输出返回(第23, 24行)。

3.4 关键类判定实验设置与结果评估

本文的实验数据来源于开源项目中的代码提交。在本节中, 我们首先介绍实验数据的收集过程, 然后介绍关键类判定模型的参数设置, 最后给出关键类判定模型的评估标准以及实验结果。

3.4.1 数据收集

我们从Sourceforge的SVN库中收集了120个开源项目的提交, 通过前文提及

算法2: 坏样本过滤算法

```

1: Input:  $N$ : The set of negative samples;  $P$ : The set of positive samples;
2:       $T$ : Total sample set,  $T = N \cup P$ ;  $C$ : Total commit set;
3: Output:  $new\_SampleSet$ ;  $new\_CommitSet$ ;
4: BadSampleOptimization ( $N, P, T, C$ ):
5:   foreach  $ps \in P$  do
6:     foreach  $ns \in N$  do
7:        $cosineSimilar \leftarrow \text{cosine}(ps.featureVector, ns.featureVector);$ 
8:       if  $cosineSimilar == 1$  do
9:          $badSample\_Set.add(ps); badSample\_Set.add(ns);$ 
10:      end if;
11:    end foreach
12:  end foreach
13:  if  $badSample\_Set != \text{null}$  do
14:    foreach  $Commit$  do
15:      if  $Commit.S \subset badSample\_Set$  do
16:         $C.remove(Commit);$ 
17:      end if
18:    end foreach
19:    foreach  $bs \in badSample\_Set$  do
20:       $T.remove(bs);$ 
21:    end foreach
22:  end if;
23:   $new\_SampleSet \leftarrow T$ ;  $new\_CommitSet \leftarrow C$ ;
24: return  $new\_SampleSet$  and  $new\_CommitSet$ ;

```

的过滤规则及坏样本过滤算法，多滤掉不满足条件的提交，最后得到4611个有效提交，我们从这4611个提交中生成实验数据。我们将在注释中提到的类标记为关键类，作为正样本集合；未提到的类标记为非关键类，作为负样本集合。在这4611个提交中，生成的数据集大小为22189条。我们按照2: 1的比例，将这些数据划分为训练集和测试集。训练集中含有3081个提交，共14780条样本。测试集中包含1530个提交，共7409条样版本。数据集统计信息如表3-6所示。

3.4.2 模型参数设置

在关键类判定中，我们选择随机森林作为模型训练算法。随机森林主要的调整参数包括：(1)基分类器个数；(2)每个基分类器的样本采样比例；(3)每个基分类器的特征采样比例。其中，基分类器个数越多，模型的分类效果越好，但与此同时模型训练的复杂度也越高。基分类器的样本采样比例越高，基分类器训练时可使用的数据也就越多，但基分类器受噪声影响的概率也越大。基分类器的特征采样比例越高，单个基分类器的分类效果越好，但各个基分类器之间的关联也就越强。我们在对这三种参数进行优化时，采用网格搜索的方式，选择在测试集中分类效果最好的一组参数作为我们的模型参数。最后的优化结果为：基分类器个数=300；基分类器的样本采样比例=30%；基分类器的特征采样比例=30%。

3.4.3 模型评估标准

本文的一个主要目标是从代码提交涉及修改的类中自动识别出关键类，并验证在识别出关键类后，对开发和维护人员理解提交中的代码修改是否有所帮助。因此，我们从两个方面对模型进行评估：

1：关键类判定模型的精确度和召回率分别为多少？

为验证我们的模型分类效果，我们主要考察两个度量指标：精确度(precision)和召回率(recall)。精确度用来衡量我们识别的关键类的准确性，召回率用来度量模型分类结果的安全性，即模型识别的关键类可覆盖实际的关键类集合的程度。同时，为了考察我们的模型在判定关键类和非关键类的综合效果，我们还度量了模型的综合准确率(accuracy)。这三个度量指标定义如下所示：

$$precision = \frac{|ActualCoreClasses \cap EstimatedCoreClasses|}{|EstimatedCoreClasses|} \times 100\% \quad (3.1)$$

表 3-6 数据集

开源项目数量	编程语言	有效提交数量	训练集提交数量	测试集提交数量	训练样本	测试样本
120	Java	4611	3081	1530	14780	7409

$$recall = \frac{|ActualCoreClasses \cap EstimatedCoreClasses|}{|ActualCoreClasses|} \times 100\% \quad (3.2)$$

$$accuracy = \frac{|CorrectJudgedClasses|}{|TotalClasses|} \times 100\% \quad (3.3)$$

其中, *ActualCoreClasses*表示数据集中真实的关键类集合; *EstimatedCoreClasses*表示被分类器分类为关键类的集合; *CorrectJudgedClasses*表示被分类器分类正确的类的数量; *TotalClasses*表示数据集中类的总数量。

我们以提交为单位衡量我们模型的分类效果时, 当且仅当提交中所有的关键类和所有的非关键类均分类正确时, 我们才认为模型正确分类了一个提交。因此, 我们选择了关键类判定的绝对准确率作为模型评估的其中一个度量标准, 其定义如下:

$$absoluteaccuracy = \frac{|CorrectJudgedCommit|}{|TotalCommit|} \times 100\% \quad (3.4)$$

其中, *CorrectJudgedCommit*表示数据集中被分类模型正确分类的提交数量; *TotalCommit*表示数据集中的提交总数。

2: 关键类的判定是否对开发和维护人员对理解代码提交有积极作用?

为了验证关键类判定方法是否对理解代码提交具有积极作用, 我们从数据集中选取了30条代码提交, 并邀请具有开发经验的程序员进行问卷调查。我们根据提交中修改类的数量对提交进行分类, 选取的修改类数量及提交个数分别为2(12条), 3(8条), 4(5条), 5(3条), 7(1条)和9(1条)个。我们针对这30个提交设置了30道问卷调查题目, 每道题目分为两种类型, 分别为包含关键类提示类型和不包含关键类提示类型。每个参与者被分配到的题目中, 包含提示与不包含提示的问题各占一半。对于每道题目, 我们给出修改类的源代码, 并且把涉及修改的代码高亮处理。参与者通过阅读代码修改及关键类提示信息(对于包含关键类信息的问题), 理解代码修改后手写出注释信息, 并记录下理解提交的开始时间和结束时间, 以及理解当前提交所花费的总时间。

我们采用本地调查和网络调查两种方式进行问卷调查。在本地调查中, 有8位志愿者参与。其中包括1位高校教师, 2位博士研究生, 以及5位硕士研究生。所有参与者均具有计算机专业的背景以及实际的开发经验。在网络调查中,

表3-7 问卷调查

问卷调查		
项目名:jEdit	版本号:24039	提交作者:dal
参与人姓名:		
开始时间:	_:_	
关键类信息:	LookAndFeelPlugin.java	
源代码:	LookAndFeelOptionPane.java LookAndFeelPlugin.java SystemLookAndFeelInstaller.java	
手写注释:		
结束时间:	_:_	
所用时间:	___(分钟)	

我们将调查问卷发布在猪八戒网^①上，并要求参与问卷调查的志愿者提供专业背景等资料。在收到的网络调查问卷中，志愿者均具有计算机专业相关的背景，且有三年及以上的Java编程经验。调查问卷的格式如表3-7所示。

在调查问卷中，我们隐藏了提交注释信息。我们要求志愿者通过阅读代码修改，编写出此次代码提交的提交注释。调查结束后，我们将代码提交的原始提交注释与志愿者编写的提交注释进行对比，如果它们表达的意思相同或者相近，我们认为该问卷为有效问卷。

最后，通过对提示关键类与不提示关键类这两类调查问卷在理解代码修改所耗费的时间，来评估关键类判定方法是否能减少代码提交的理解时间；同时，通过比较提示关键类与不提示关键类这两种调查方式产生的有效调查问卷的数量，考察关键类判定方法是否能提高代码提交的理解准确率。

3：关键类的判定在代码评审中是否具有积极作用？

关键类的判定在现实项目的代码评审中是否具有积极作用？为了验证这个问题，我们选择了5个开源项目，分别是：*Eclipse CDT*, *Eclipse JDT Core*, *Eclipse Platform UI*, *OpenDaylight Controller*以及*Vaadin*。然后对这些开源项目的代码评审人员进行了问卷调查。问卷包含11个问题，其中的10个问题是关于代码评审人员的背景以及开发经验，剩余的一个问题为关键类判定的准确率评

^① 猪八戒网,<http://www.zbj.com>,2018

估。最后，我们一共收回了16份问卷结果。通过分析调查问卷的结果，评估关键类判定方法在代码评审中是否具有积极作用。

3.4.4 关键类判定模型实验结果

1：关键类判定模型的精确度和召回率分别为多少？

表 3-8 判定结果

提交数量		样本数量		正确分类的样本数量		错误分类的样本数量	
正确判定的提交数量	错误判定的提交数量	正样本(1)	负样本(0)	正确分类为1	正确分类为0	错误分类为1	错误分类为0
1127	403	1714	5695	1285	5266	429	429
合计： 1530		合计： 7409		合计： 6551		合计： 858	

分类模型在测试集中的分类结果如表3-8所示。测试集中的提交数量总共为1530个，其中，1127个被分类正确，403个被分类错误。在这1530个提交中，总共包含7409个涉及修改的类，其中，关键类为1714个，非关键类为5695个。被正确分类的关键类个数为1285，正确分类为非关键类的个数为5266。被错误分类为关键类的个数为429，被错误分类为非关键类的个数为429。

表 3-9 精确度与召回率

样本类型	precision	recall
正样本	74.97%	74.97%
负样本	92.47%	92.47%

表3-9为分类模型的精确度和召回率结果。其中，对于关键类，精确度为74.97%，召回率为74.97%；对于非关键类，精确度为92.47%，召回率为92.47%。从表3-9中可以发现，我们的分类模型进行关键类和非关键类的分类时，结果差异较大。这主要是因为正负样本的不平衡造成的。在数据集中，正负样本的比例大概为1: 3，原本为负样本的样本被分类器误分类为正样本后(或者原本为正样本的样本被分类器误分类为负样本后)，在正负样本总数不变的前提下，分类器的分类结果的精确度和召回率的下降幅度的比值大约为3: 1。

表3-10和表3-11分别为分类模型的综合准确率和绝对准确率的结果。从表3-10中，可以看到分类模型在关键类和非关键类的分类结果的综合准确率达

表 3-10 综合准确率

TotalClasses	CorrectJudgedClasses	accuracy
7409	6551	88.42%

表 3-11 绝对准确率

TotalCommit	CorrectJudgedCommit	absoluteaccuracy
1530	1127	73.66%

到88.42%，表示分类模型可以正确的分类出大部分的关键类和非关键类。从表3-11的结果看到，在以提交为单位时，得到的绝对准确率为73.66%。

2：关键类的判定是否对开发人员理解代码提交有积极作用？

我们通过本地调查和网络调查两种方式进行问卷调查。最后一共收到了218份问卷。统计结果如图3-1(a)所示。其中，带关键类信息的有效问卷为103份，不带关键类信息的有效问卷为86份，有效问卷总数为189份；带关键类信息的无效问卷为6份，不带关键类信息的无效问卷为23份，无效问卷总数为29份。

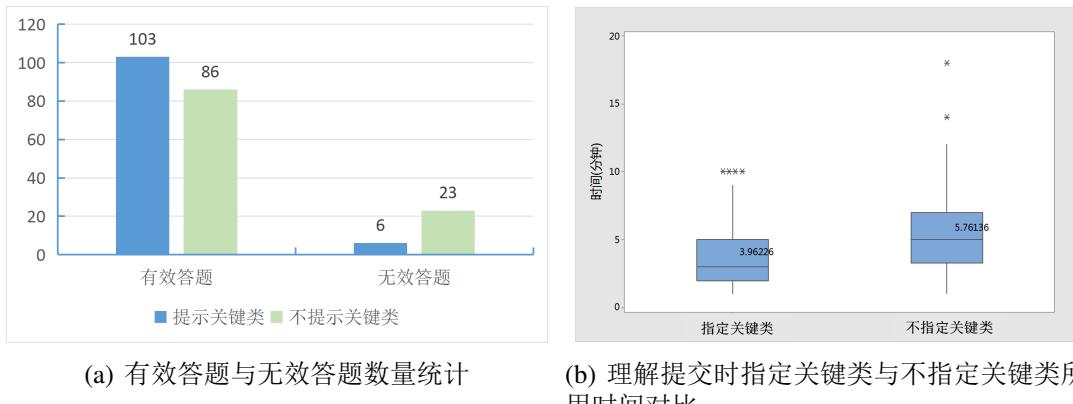


图 3-1 问卷调查结果统计

从问卷答题数量的统计结果中可以看出，在有效答题的问卷中，带关键类提示信息的问卷比不带关键类的问卷多17份；在无效答题的问卷中，带关键类提示信息的问卷比不带关键类提示信息的问卷少17份。说明了通过提示关键类信息，可以提高开发人员理解代码提交的准确度。与此同时，我们还对参与问卷调查的志愿者进行了回访。在进行问卷调查过程中，有一些志愿者感受到：“通过提示关键类信息，可以帮助开发人员把握代码修改的整体逻辑”，还有一

些人觉得：“在代码提交中，常会由于关键类的修改，而引起非关键类的修改”；另外，还有一些人觉得“在没有关键类信息的情况下，也可通过对代码修改的理解推测出关键类，只是相比于有关键类信息的情况下，需要花费更多的时间”。综上所述，从问卷调查的结果统计以及对志愿者的回访结果中可以得出一个结论：关键类判定方法可以有效地辅助开发人员理解代码提交。

此外，我们还对关键类判定方法是否可以缩短开发人员理解代码提交的时间进行了研究。我们将有效答题问卷分为两部分，分别为带关键类信息的问卷以及不带关键类信息的问卷。并对这两类问卷中对代码提交的理解时间进行了统计。如图3-1(b)所示，带关键类提示的有效问卷的答题时间平均为3.96分钟；不带关键类提示的有效问卷的答题时间平均为5.76分钟。从该统计结果可以看出，通过提供关键类信息，每个代码提交的理解时间平均可以缩短1.8分钟。且图3-1(b)还显示，带关键类信息的问卷最长答题时间为10分钟，最短为1分钟；不带关键类信息的问卷最长答题时间为18分钟，最短为1分钟。为了验证提示关键类与不提示关键类对代码提交的理解时间是否有显著性差异，我们采用Wilcoxon假设检验方法[?]检验如下的空假设：

假设 H_0 ：提示关键类与不提示关键类对代码提交的理解时间没有显著性差异。

表 3-12 假设检验结果

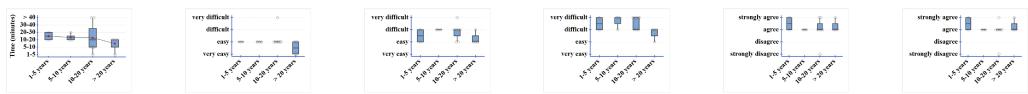
点估计值	95% 置信区间	w	p
-2.0	(-2.0,-1.0)	8546.5	<0.0001

检验结果如表3-12所示，检验统计量w等于8546.5的p值在对结调整时小于0.0001，由于p值小于所选水平0.05，因此有足够的证据否定原假设 H_0 ，认为提示关键类与不提示关键类对代码提交的理解时间具有显著性差异。因此，我们可以得出结论：关键类信息对于代码提交的理解具有重要作用。

3:关键类的判定在代码评审中是否具有积极作用?

我们从5个开源项目的代码评审人员中，共收回16份问卷调查结果。首先，为了验证关键类的判定在代码评审中是否具有积极作用，我们从这5个开源项目中随机选择最近两个月的代码提交，并使用关键类判定方法识别出代码提交中的关键类。然后，我们将这个代码提交及关键类信息加入到调查问卷中。在

调查问卷中，我们提问代码评审人员，在问卷中给出的关键类是否为代码提交的关键类。从收到的16份问卷调查结果中，有14人认为关键类判定方法识别的关键类是代码提交中的关键类，有2人认为方法识别的关键类不是代码提交的关键类。如在其中一个参与者的问卷结果：代码提交的关键类不是方法识别的*OpenSshConfigTest*，而是类*OpenSshConfig*。我们通过查看这两个类的源代码发现，这两个类没有直接的关系，也就是说，这个代码提交并非是原子提交，它们中不存在关键类。



(a) 代码评审 (b) 代码提交 (c) 5 ≤ 代码提交的类 < 5个 (d) 代码提交的类 ≥ 10 的类的重要性 (e) 阅读顺序 (f) ICC方法有助于代码评审

图 3-2 问卷调查结果

另外，我们还对参与我们问卷调查的代码评审人员作了回访，并提了几个问题，第一个问题是平均一个代码提交所花费的评审时间为多少？结果如图3-2(a)所示，其中有一半的参与者的答案在10-20分钟之间，有两个参与者的答案大于40分钟，另外6个参与者的答案在1-10分钟之间。我们发现需要40分钟评审一个代码提交的两个参与者，其所在的项目是*Eclipse CDT*和*Eclipse JDT Core*，这两个项目的代码提交涉及的类的修改范围都比较大。我们还发现评审时间还随着参与者的开发经验的增加而减少。

第二个问题是代码评审难度与代码提交中的修改类个数是否相关。结果如图3-2(b)，3-2(c)和3-2(d)所示，当代码提交涉及的类小于5个时，评审人员普遍认为这个代码提交是容易理解的。而当代码提交的类在5个到10个之间时，将近一半的评审人员认为代码提交不易于理解。当代码提交的类为10个及以上时，超过90%的评审人员认为代码提交不易于理解。另外，评审人员还认为代码提交的理解不仅受到代码提交的类的个数的影响，还受到类的修改范围等的影响。

第三个问题是合理的阅读顺序是否有助于代码提交的理解。结果如图3-2(e)所示，16个评审人员中，有15个认为合理的阅读顺序有助于代码提交的理

解，只有一个评审人员持怀疑态度，他认为合理的阅读顺序与按照代码修改行数从多到少阅读代码对代码提交的理解并无太大不同。但实际上，在很多时候代码修改行数并不能很好的衡量修改类的重要性，如：在一个类中多处调用了一个修改的方法，如果从代码修改行数的多少来衡量类的重要性，那么会认为涉及方法调用的类对理解代码提交更为重要。然而，实际情况是因为方法实现的变化，才引起了方法调用的变化。所以，修改的方法所在的类才是代码提交的关键类。

最后一个问题是关键类判定方法是否有助于代码提交的代码评审。从图3-2(f)中可以看出，有15个评审人员认为关键类判定方法可以帮助评审人员对代码提交进行代码评审，只有一个评审人员持反对态度，而该评审人员与在第三个问题中持怀疑态度的是同一个人。我们的猜测是该评审人员或许没有很好的利用好关键类信息，而是按照他原有的方式对代码提交进行代码评审。

综上所述，从调查问卷的结果可以看出，代码评审一项费时费力的工作，代码评审的复杂度与代码提交的类的数量及类的代码修改复杂度关系密切。且大多数评审人员认为合理的阅读顺序有助于代码提交的理解，而通过提供关键类的信息，有助于提高代码评审的效率。

3.4.5 关键类判定影响因素讨论

在实验中，我们的数据来源为开源项目的代码提交数据。这些代码提交数据质量参差不齐，为了保证数据质量，我们设置了多种数据过滤规则，包括：(1)过滤掉不含代码修改的代码提交；(2)过滤掉没有提交注释的代码提交；(3)过滤掉没有在提交注释中提及修改的类的代码提交。在经过过滤之后，有效的代码提交数量为4611个。在进行关键类判别模型训练时，训练集相对较小。这有可能会影响模型的性能。在数据集的采集过程中，我们将提交注释中提到的类作为关键类，未提到的类作为非关键类。这在大多数时候可有效地对数据添加标签，但也有可能存在一些噪声。另外，在我们采集的数据集中，正负样本比例为1:3。数据集的不平衡性也有可能影响到模型的性能。

3.5 本章小结

代码提交是软件工程过程中的重要数据。帮助开发人员快速理解代码提

交，可以有效提高软件维护的效率。本文将代码提交中涉及修改的类按照其重要程度的不同分为关键类与非关键类。关键类信息有助于开发人员理解代码提交。为了自动识别代码提交中的关键类，本文提出了一种基于机器学习的关键类判别方法。该方法从代码提交中提取可判别特征，并使用带监督的机器学习算法进行建模和评估。实验结果表明，该方法判定关键类的综合准确率达到了88.4%。且在研究问题二与问题三中，验证了使用关键类判定方法可以帮助开发和评审人员理解代码提交。

第4章 代码修改分析与注释检查系统

本章结合第二章和第三章的内容，将注释一致性检测方法和关键类判定方法引入到我们的代码修改分析和注释检查系统中。以下将详细介绍系统的具体实现。其中，该系统主要分为三大模块：(1)代码修改分析模块；(2)注释检查模块；以及(3)基础服务模块。其中，代码修改分析模块中的关键类检测功能和注释检查模块中的注释一致性检测功能为系统的核心功能，其他功能模块为这两个核心功能的辅助模块。系统的实现采用MVC(Model-View-Controller)模式，实现平台为Java，采用Spring Boot框架进行系统构建，前端使用Thymeleaf引擎，后台数据库为MongoDB数据库。系统功能结构图如图4-1所示。

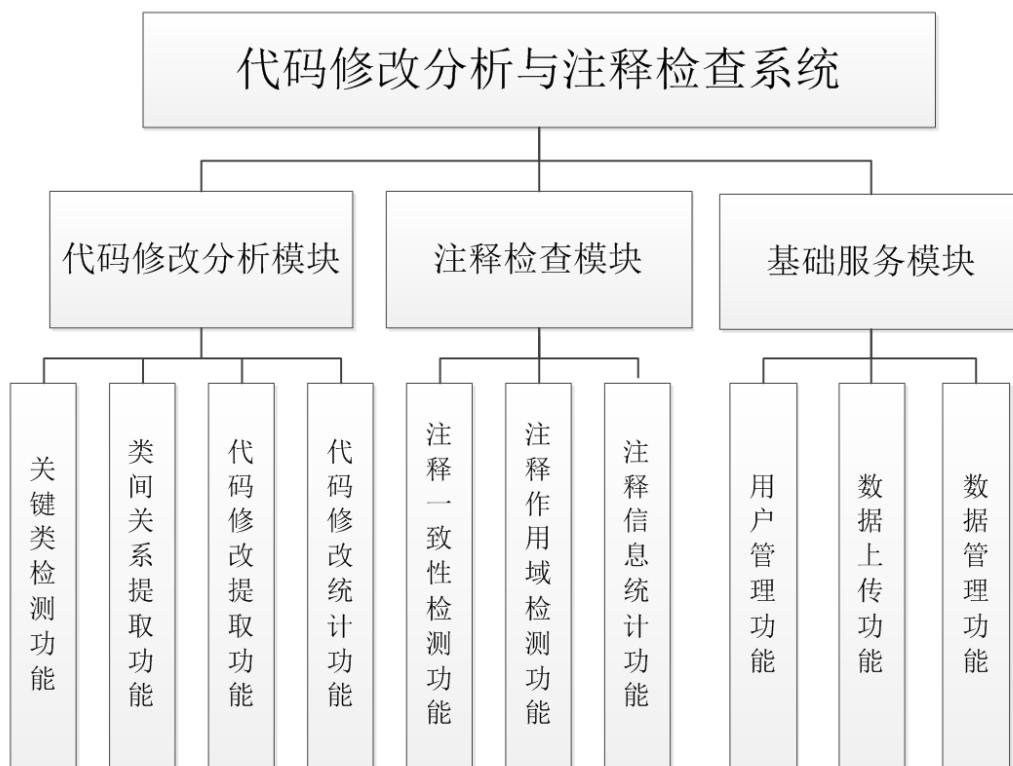


图 4-1 系统功能结构图

4.1 代码修改分析模块

代码修改分析模块包括以下几个功能：(1)关键类检测功能；(2)类间关系提取功能；(3)代码修改提取功能；(4)代码修改统计功能。该模块主要完成关键类

的检测与展示，类间关系的提取与展示，代码修改的提取与展示以及每个类的修改信息的统计与展示等任务。其中，关键类检测功能为本模块的核心功能。以下我们将对每个功能作简要的阐述。

4.1.1 关键类检测功能

关键类检测功能作为我们的系统的核心模块之一，其主要负责对用户上传的修改类文件作关键类判定。该功能主要包括两方面的内容：(1)关键类分类模型的构建；(2)关键类分类模型的分类结果展示。其中，关键类分类模型的构建过程如图4-2所示。我们从开源代码库中爬取项目提交信息，获取到涉及修改的源代码文件，并通过抽象语法树解析得到代码修改信息。然后根据第三章所描述的特征提取方法提取特征，得到训练特征向量列表，用于关键类分类模型的训练。在关键类分类模型训练完成后，我们对用户上传的数据也进行类似的过程，得到待分类特征向量列表。最后经过关键类分类模型的分类，输出分类结果。至此，我们实际上已经得到了用户上传的修改类作为关键类的概率值(0-1之间的小数值)。

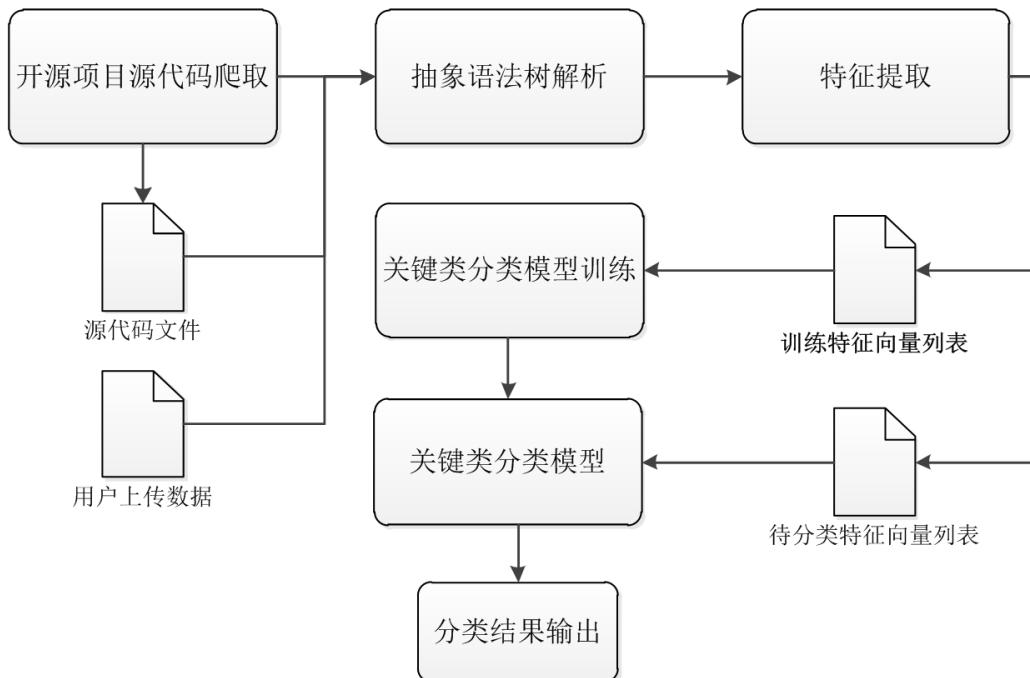


图 4-2 关键类分类模型构建过程

在对关键类分类模型的分类结果进行展示时，我们采用鱼网图的方式。在鱼网图中，修改类作为关键类的概率值越大，其所占面积也越大。换句话说，

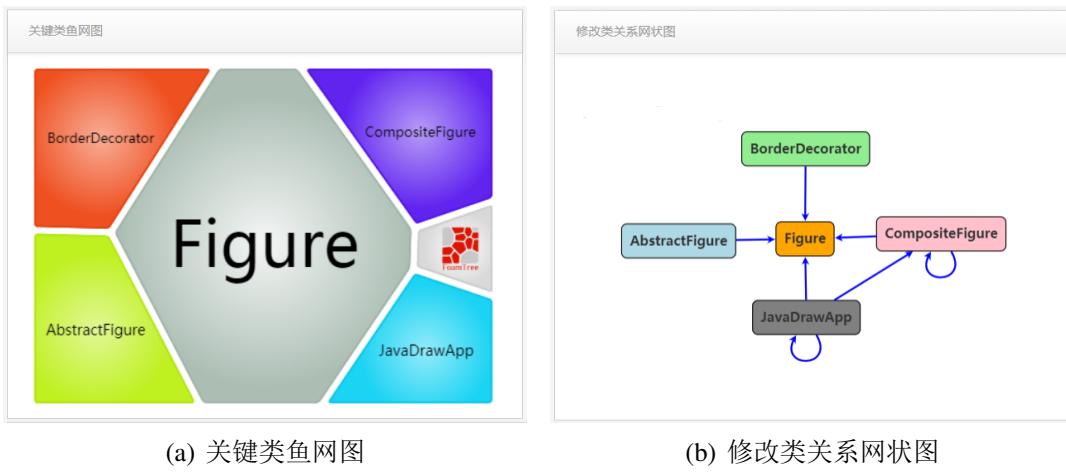


图 4-3 关键类鱼网图和修改类关系网状图

该类在此次修改中也就显得越重要。展示效果如图4-3(a)所示，该图说明用户此次上传的文件中涉及5个修改类。在这5个修改类中，*Figure*类所占面积最大，即它作为关键类的概率越大。这说明了此次修改*Figure*类为主要修改的类，开发和维护人员在进行代码理解时，应主要关注*Figure*类的变化。

4.1.2 类间关系提取功能

类间关系提取功能主要负责提取修改类之间的调用关系，并以可视化方式展现给用户。通常而言，在一次代码修改中，修改类之间并不互相独立，它们或多或少存在着一些耦合关系。我们在对用户上传的数据进行抽象语法树解析的同时，也提取了每个修改类之间的调用关系，并且在我们的系统中采用网状图的形式展现出来。展示效果如图4-3(b)所示。从图中可以看出，*Figure*类均被其他类使用到。而从我们的关键类鱼网图中亦可看出，*Figure*类为主要修改的类。在这里再次印证了我们将*Figure*类作为关键类的猜想。

4.1.3 代码修改提取功能

代码修改提取功能作为以上三个功能的基础服务模块存在，其主要负责对用户上传的文件进行新旧版本比对，并通过抽象语法树提取代码变化实体，如变化的类，变化的方法，变化的语句等。同时，我们还实现了新旧文件基于行的比对。其主要用于用户上传的源代码的展示，以方便开发和维护人员阅读代码及其修改的部分。结合核心类检测功能，我们将修改类按照其关键类概率从大到小依次展示其源代码，且高亮显示涉及代码变化的行。其中，新增行在行

开头增加了“+”标记，删除行在行开头增加了“-”标记。在图4-4中给出了完整示例，包括关键类鱼网图，修改类关系网状图，修改类摘要信息以及源代码展示(由于页面长度限制，省略了关键类概率值排名倒数前三的修改类的展示)。

4.1.4 代码修改统计功能

代码修改统计功能负责统计修改类的相关变化信息。其展示页面如图4-5所示。其中，数字1标示部分通过表格形式展示修改类的信息。展示内容包括类编号，类名，旧类代码行，新类代码行，旧类方法数，新类方法数以及类的变化类型。数字2至数字5标示部分通过柱状图的形式分别对类的变化方法个数，变化语句个数，类的入度以及类的出度进行展示。其中，变化方法个数以及变化语句个数又包括新增，修改和删除三种类型，在柱状图中我们通过不同颜色对其进行区分。类的入度指的是该类被多少个修改类引用过，类的出度表示该类引用的修改类个数。我们通过数字1表示的表格可以准确的知道每个类的修改信息，而从数字2至数字5所示的柱状图中，则可以对每个类的修改幅度以及引用其他类和被其他类引用的多少作出直观和清晰的比较。通过结合关键类检测功能以及类间关系提取功能，可以更准确且快速地把握主要修改的类，从而提高开发和维护人员理解代码修改的效率。

4.2 注释检查模块

注释检查模块包括以下几个功能：(1)注释一致性检测功能；(2)注释作用域检测功能；(3)注释信息统计功能。该模块主要完成注释一致性的检测与展示，注释作用域的检测与展示，以及修改类中注释信息的统计与展示等任务。其中，注释一致性检测功能为该模块的核心功能。以下我们将对这几个功能作简要的说明。

4.2.1 注释一致性检测功能

注释一致性检测功能是系统的一个核心功能，其主要负责对用户上传的修改类文件中的代码和注释作一致性检查。该功能的流程图如图4-6所示。

首先，我们从开源代码库中爬取训练数据，通过抽象语法树解析得到修改的代码片段，并利用注释作用域检测功能将注释和代码关联。然后，根据第二

章所描述的特征提取方法提取特征，得到训练特征向量集，用于注释一致性分类模型的训练。在分类模型训练完成后，我们对用户上传的数据也进行类似的过程，得到待检测特征向量集。最后通过分类模型的分类，得到分类结果，并将分类结果为与代码不一致的注释及其相关代码提交给用户以供查看和修复。最终检测结果如图4-7所示。其中，数字1表示部分为注释一致性信息表格，表格内容包括注释编号，注释所属类，注释作用域范围和注释与代码的一致性。注释信息按照与代码不一致的概率从大到小排列。数字2标示部分为注释及与注释相关的代码信息。代码片段中涉及修改的代码高亮显示，以提示用户重点关注变化的代码。

4.2.2 注释作用域检测功能

注释作用域检测功能主要负责确定注释所关联的代码的范围。系统根据第二章中算法1所示的算法完成注释作用域的检测。然后，在代码显示区域只显示在注释作用域范围内的代码。从而达到关注与注释密切相关的代码片段，而忽略掉那些与注释关系不大的代码的效果。如图4-7所示，数字2标示部分为注释及与注释相关的代码信息，其忽略了在注释作用域范围外的代码。

4.2.3 注释信息统计功能

注释信息统计功能负责统计修改类中的注释信息。其展示页面如图4-8所示。其中，数字1标示部分通过表格形式展示类中注释的信息。展示内容包括类编号，类名，代码行，注释行，文档注释个数，块注释个数以及行注释个数。数字2至数字5标示部分通过柱状图的形式分别对类中的注释密度，注释长度，注释的语义类型以及注释的语法类型个数进行展示。其中，注释的语义类型包括“TODOs”，“Bug”，“Note”和“Common”四种类型。注释的语法类型包括文档注释，块注释和行注释类型。我们通过数字1标示的表格可以准确的知道每个类中的注释信息，而从数字2至数字5所示的柱状图中，则可以对每个类中的注释情况作出直观清晰的比较。结合这两类信息以及注释一致性信息，用户可以对类中的注释质量作出一个大致的判断。

4.3 基础服务模块

基础服务模块包括以下几个功能：(1)用户管理功能；(2)数据上传功能；(3)数据管理功能。其中，用户管理功能主要完成用户的注册，登录，注销等操作。数据上传功能主要完成用户的代码修改上传，分析和保存等操作。数据管理功能主要完成用户历史上传数据的搜索，查看和删除等操作。

4.4 本章小结

在本章中，我们简要地介绍了代码修改分析和注释检查系统的功能实现，其主要包含三个功能模块，分别为：代码修改分析模块，注释检查模块和基础服务模块。代码修改分析模块集成了第三章中关键类判定的相关内容，注释检查模块集成了第二章中代码和注释一致性检测的相关内容，基础服务模块主要是为用户及其相关数据管理服务的模块。通过代码修改分析和注释检查系统，用户可以在线分析一次代码提交的代码修改，并检查代码和注释的一致性，从而提高代码提交的质量。

Change Analysis

文件处理 : No file selected Choose File 上传

核心类分析

关键类鱼网图

修改类关系网状图

类修改摘要信息

关键类:Figure

新增方法: drawAll; drawDecorators; addFigureDecorator; removeFigureDecorator; figureDecorators;

非关键类:CompositeFigure
修改方法: draw;

非关键类:AbstractFigure
新增方法: addFigureDecorator; removeFigureDecorator; figureDecorators; drawAll; drawDecorators;
修改方法: init; write; read;

非关键类:BorderDecorator
修改方法: BorderDecorator; BorderDecorator;
删除方法: basicDisplayBox; handles; getDecoratedFigure;

Figure

```

83     /**
84      * Draws the figure.
85      * @param g the Graphics to draw into
86      */
87     public void draw(Graphics g);
88
89     /**
90      * This method will call draw first the drawDecorators
91      */
92     public void drawAll(Graphics g);
93
94     /**
95      * This method draws the FigureDecorators
96      * @see FigureDecorator#draw
97      */
98     public void drawDecorators(Graphics g);
99
100    /**
101     * Returns the handles used to manipulate
102     * the figure. Handles is a Factory Method for
103     * creating handle objects.
104     *
105     * @return an type-safe iterator of handles
106     * @see Handle
107     */
108    public HandleEnumeration handles();
109
110    /**
111     * Gets the size of the figure
112     */
113    public Dimension size();
114
115    /**
116     * Gets the figure's center
117     */
118    public Point center();
119
120    /**
121     * Checks if the Figure should be considered as empty.
122     * For instance, when the figure is being added to the drawing, the <code>
123     * CreationTool</code> will check this to see whether to add the figure or
124     * not. If the figure is too small to be seen, it will return true for isEmpty
125     * under the above condition.
126     */
127    public boolean isEmpty();
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470

```

ComposedFigure

```

425     /**
426      * if we are asked to draw figures that we do not contain, Exception?
427      * @see Figure#draw
428      */
429     public void draw(Graphics g, FigureEnumeration fe) {
430         while (!fe.hasMoreFigure()) {
431             fe.nextFigure().draw(g);
432             fe.nextFigure().drawAll(g);
433         }
434     }
435
436     /**
437      * Gets a figure at the given index.
438      */
439     public Figure figureAt(int i) {
440         return (figure) getFigures().get(i);
441     }
442
443
444     /**
445      * Returns an enumeration for accessing the contained figures.
446      * The enumeration is a snapshot of the current contained <b>Figure</b>s
447      * and is not a "live" enumeration and does not take subsequent
448      * changes of the <b>CompositeFigure</b> into account.
449      * The figures are returned in the drawing order.
450
451
452     public FigureEnumeration figures() {
453         return new FigureEnumerator(CollectionsFactory.current().createList(getFigures()));
454     }
455
456     /**
457      * Returns an enumeration to iterate in
458      * Z-order back to front over the (glink Figure Figure)s
459      * that lie within the given bounds.
460
461     public FigureEnumeration figures(Rectangle viewRectangle) {
462         if (_theQuadTree != null) {
463             FigureEnumeration fe =
464                 _theQuadTree.getAllWithin(new Bounds(viewRectangle).asRectangle2D());
465             List l2 = CollectionsFactory.current().createList();
466             while (fe.hasMoreFigure()) {
467                 Figure f = fe.nextFigure();
468             }
469         }
470     }

```

图 4-4 修改类信息展示页面

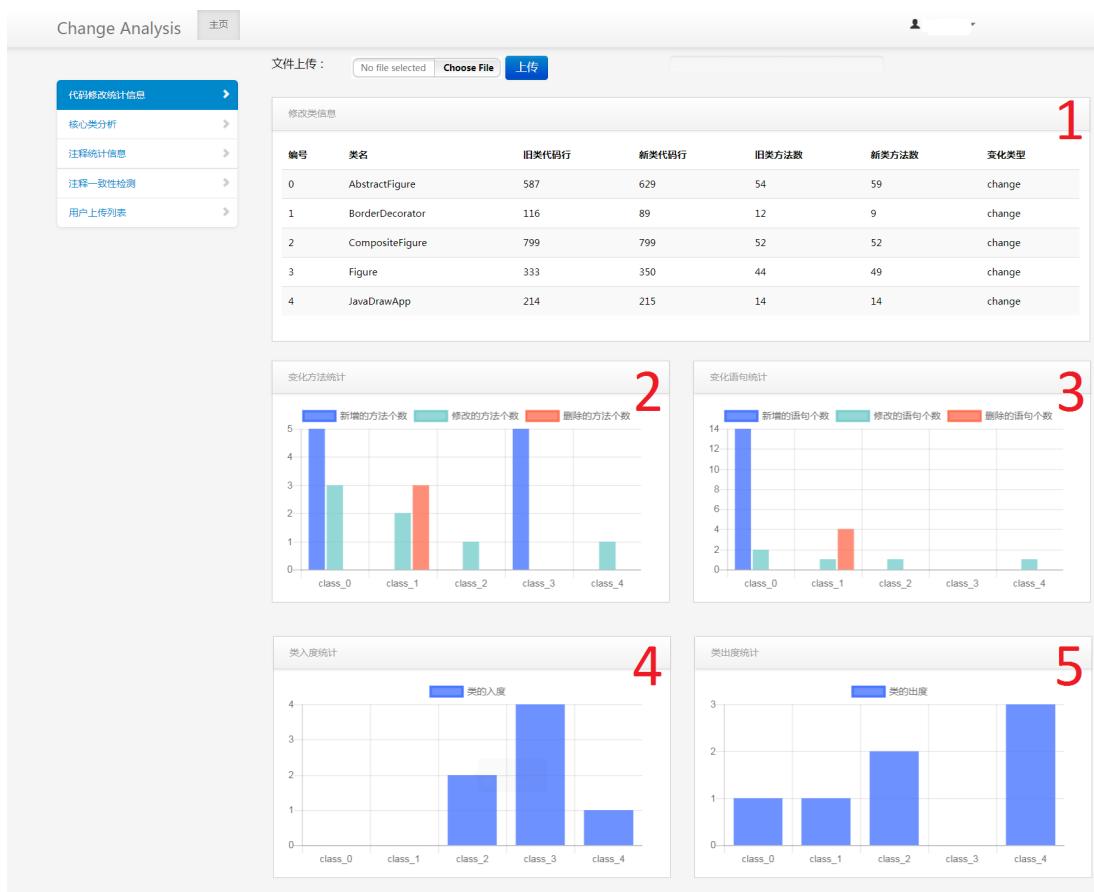


图 4-5 代码修改统计页面

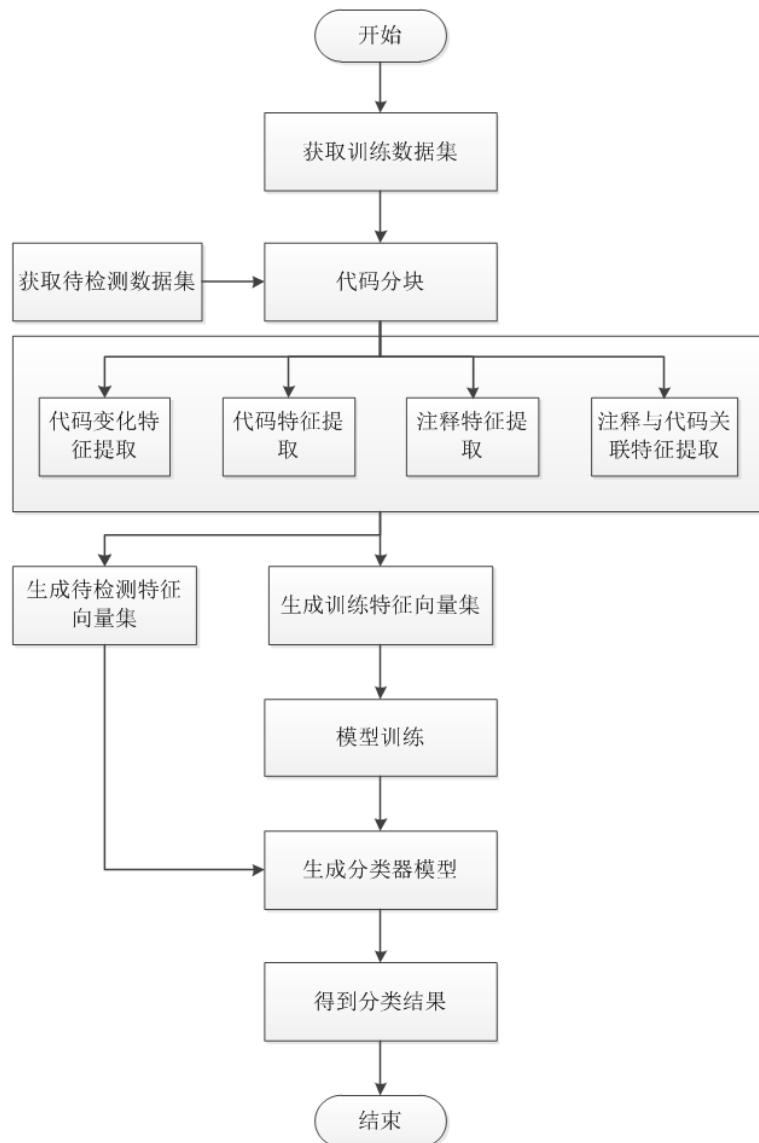


图 4-6 注释一致性检测流程图

Change Analysis

文件上传 : No file selected Choose File 上传

编号	所属类	范围	与代码的一致性	查看
2	AbstractFigure	506-518	不一致	查看
5	BorderTool	79-86	不一致	查看
0	AbstractFigure	89-93	一致	查看
1	AbstractFigure	478-493	一致	查看
3	BorderDecorator	93-97	一致	查看
4	BorderTool	50-61	一致	查看
6	CompositeFigure	333-342	一致	查看

AbstractFigure comment_2

```
506 //load figureManipulators
507 int manipSize = dr.readInt();
508 ffigureManipulators = CollectionsFactory.current().createList(manipSize);
509 for (int i=0; i<size; i++) {
510     ffigureManipulators.add((FigureManipulator)dr.readStorable());
511 }
512
513 int decSize = dr.readInt();
514 ffigureDecorators = CollectionsFactory.current().createList(decSize);
515 for (int i=0; i<size; i++) {
516     ffigureDecorators.add((Figure)dr.readStorable());
517 }
518 }
```

BorderTool comment_5

```
79 //TODO: Peels off the border from the clicked figure.
80 UndoActivity.createUndoActivity();
81 list1 = CollectionsFactory.current().createList();
82 l.add(figure);
83 l.add((DecoratorFigure)figure).peelDecoration();
84 getUndoActivity().setAffectedFigures(new FigureEnumerator(l));
85 ((BorderTool.UndoActivity)getUndoActivity()).replaceAffectedFigures();
86 }
87 }
```

AbstractFigure comment_0

```
89 // TODO: may have more entry to init.
90 mDependentFigures = CollectionsFactory.current().createList();
91 ffigureManipulators = CollectionsFactory.current().createList();
92 ffigureDecorators = CollectionsFactory.current().createList();
93 }
```

图 4-7 注释一致性检测页面

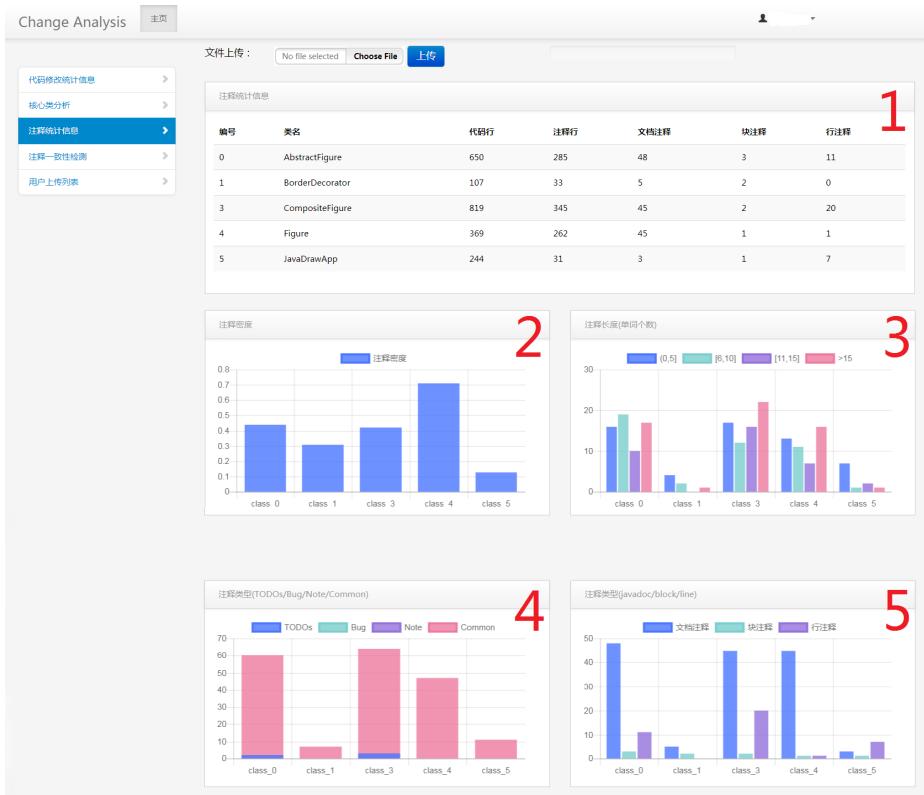


图 4-8 注释信息统计页面

第5章 总结与展望

代码提交作为软件演化过程中的重要产物，对开发和维护人员理解代码修改和进行软件维护具有重要意义。在代码提交中包含提交注释，修改代码，与修改代码相关的代码注释以及其他信息。帮助开发和维护人员快速有效地分析和理解这些信息，将有助于软件的质量控制和维护。本研究的目的是为了帮助开发和维护人员在软件维护过程中快速高效地理解和评估代码提交。本章将对前文工作进行总结，并对未来的研究工作进行展望。

5.1 工作总结

代码提交在软件维护活动中扮演着极其重要的角色，本文针对代码提交的评估和理解问题展开研究。首先，在代码提交的评估方面，我们提出了数据驱动的代码和注释一致性检测方法，通过检测在代码修改过程中代码和注释的一致性，评估代码提交的质量。其次，在代码提交的理解方面，我们提出了基于关键类判定的代码提交理解辅助方法，通过判定在代码提交中哪些类是被主要修改的类，以引导开发人员正确地阅读代码修改，达到快速高效地理解代码提交的目的。

代码提交的质量对开发和维护人员理解代码修改有着重要的影响。本文针对在代码修改中代码和注释的一致性评估问题，提出一种数据驱动的代码和注释一致性检测方法。该方法从代码提交中提取涉及修改的代码片段及其相关联的注释，从多个维度提取可判别特征，并按照机器学习的方法进行建模和评估。实验中我们从5个开源项目9909个代码提交中提取了35050个软件变化，在代码修改过程中，与代码不一致的注释为4595个，一致的注释为30455个。我们根据这些数据建立了一个有监督的分类模型。模型评估结果表明，在对与代码不一致的注释的分类上，准确率达到77.2%，召回率达到74.6%；对与代码一致的注释分类上，准确率达到96.4%，召回率达到97.2%。另外，我们选取了2000个不在模型的训练集和测试集中的软件变化，利用我们的分类模型进行分类，共找到了241个与代码不一致的注释。其中，有31个注释在原项目中未作出修改。该

实验表明我们的代码和注释一致性检测模型可有效地帮助开发和维护人员在代码提交中检测与代码不一致的注释，辅助开发和维护人员评估代码提交的质量。

正确引导开发和维护人员理解代码提交，有助于提高软件维护的效率。本文以此为目的提出一种基于关键类判定的代码提交理解辅助方法。该方法从代码提交中提取可判别特征，并使用带监督的机器学习算法进行建模和评估。实验中我们从120个开源项目中收集了4611个代码提交，共22189个修改类。实验结果显示，我们的模型判断关键类的综合准确率达到了88.4%，绝对准确率达到了73.6%。且在我们的问卷调查结果中显示，使用关键类判定方法，可以帮助开发和评审人员理解代码提交。

最后，我们将代码和注释一致性检测方法和关键类判定方法集成到我们的代码修改分析与注释检查系统中。该系统以这两个方法为核心，结合多种数据可视化方法辅助开发人员理解和分析代码修改。

综上所述，本文从代码提交的评估和理解两个方面展开了相关的研究，提出了数据驱动的代码和注释一致性检测方法和基于关键类判定的代码提交理解辅助方法，并实现了一个代码修改分析与注释检查系统。我们希望本文的研究工作可在实践中有效地辅助开发和维护人员理解和分析代码提交，并且能够为将来的关于代码提交的研究工作提供一些参考。

5.2 研究展望

本文提出的数据驱动的代码和注释一致性检测方法以及基于关键类判定的代码提交辅助理解方法，经过实验验证，结果表明可有效帮助开发和维护人员评估和理解代码提交。与此同时，这两个方法也还存在着一些不足之处，有很多问题需要在未来工作中作进一步研究，包括：

代码和注释一致性检测方面：(1)优化模型的训练数据的质量。代码和注释一致性检测模型的训练数据直接采集自开源项目，在训练集中不可避免地包含一部分噪声，这些噪声可能会影响模型的准确率。对训练数据进行筛选和验证，以增加模型的可靠性。(2)改进注释作用域检测算法。在进行数据提取时，我们采用启发式规则检测注释作用域。在一些情况下，启发式规则倾向于扩大注释

的作用域。这时有可能将一些与当前注释不相关的代码包含进来，从而影响我们的分类模型的准确率。通过进一步改进注释作用域检测算法，提高作用域检测的精确度，以提高模型的准确率。(3)改进模型的分类算法。本文使用的代码和注释一致性检测的算法是基于传统的机器学习算法，准确率还未达到理想值。通过研究利用深度学习替代传统机器学习算法，来进一步提高检测模型的准确率。(4)增加检测模型的多语言支持。由于我们在实验中选择的项目编程语言均为Java，因此，我们的分类模型可能不能很好地适应其他编程语言类型的项目。通过增加其他类型的语言的支持，扩大分类模型的适用范围。

代码提交关键类判定方面：(1)增大模型的训练集。在判定模型的数据集中，我们采集的数量还远未达到饱和状态，可进一步在开源项目中进行代码提交数据的采集。(2)改进模型的分类算法。关键类判定模型采用的分类算法为传统的机器学习算法，未来可将深度学习引入到判定模型中，来进一步提高判定模型的准确率。(3)增加更丰富的提示信息。本文提出的基于关键类的代码提交理解辅助方法，提供给用户的辅助信息比较单调，未来可增加更丰富的提示信息，如：类之间的修改传递信息，类的代码修改摘要信息，多次提交间的代码修改追踪信息等。

References

- [1] C. Szabo, “Novice code understanding strategies during a software maintenance assignment,” in *Proceedings of the 37th International Conference on Software Engineering-Volume 2.* IEEE Press, 2015, pp. 276–284.
- [2] P. Rovegård, L. Angelis, and C. Wohlin, “An empirical study on views of importance of change impact analysis issues,” *IEEE Transactions on Software Engineering*, no. 4, pp. 516–530, 2008.
- [3] S. Kim and M. D. Ernst, “Prioritizing warning categories by analyzing software history,” in *Proceedings of the Fourth International Workshop on Mining Software Repositories.* IEEE Computer Society, 2007, p. 27.
- [4] A. W. Bradley and G. C. Murphy, “Supporting software history exploration,” in *Proceedings of the 8th working conference on mining software repositories.* ACM, 2011, pp. 193–202.
- [5] M. Wittenhagen, C. Cherek, and J. Borchers, “Chronicler: Interactive exploration of source code history,” in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems.* ACM, 2016, pp. 3522–3532.
- [6] M. D’ Ambros, H. Gall, M. Lanza, and M. Pinzger, “Analysing software repositories to understand software evolution,” in *Software evolution.* Springer, 2008, pp. 37–67.
- [7] S. W. Thomas, B. Adams, A. E. Hassan, and D. Blostein, “Studying software evolution using topic models,” *Science of Computer Programming*, vol. 80, pp. 457–479, 2014.
- [8] G. Gousios, E. Kalliamvakou, and D. Spinellis, “Measuring developer contribution from software repository data,” in *Proceedings of the 2008 international working conference on Mining software repositories.* ACM, 2008, pp. 129–132.

- [9] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, “Social coding in github: transparency and collaboration in an open software repository,” in *Proceedings of the ACM 2012 conference on computer supported cooperative work*. ACM, 2012, pp. 1277–1286.
- [10] G. Canfora and L. Cerulo, “Impact analysis by mining software and change request repositories,” in *Software Metrics, 2005. 11th IEEE International Symposium*. IEEE, 2005, pp. 9–pp.
- [11] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. L. Collard, “Blending conceptual and evolutionary couplings to support change impact analysis in source code,” in *Reverse Engineering (WCRE), 2010 17th Working Conference on*. IEEE, 2010, pp. 119–128.
- [12] X. Sun, B. Li, H. Leung, B. Li, and Y. Li, “Msr4sm: Using topic models to effectively mining software repositories for software maintenance tasks,” *Information and Software Technology*, vol. 66, pp. 1–12, 2015.
- [13] E. Wong, T. Liu, and L. Tan, “Clocom: Mining existing source code for automatic comment generation,” in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 2015, pp. 380–389.
- [14] Y. Huang, Q. Zheng, X. Chen, Y. Xiong, Z. Liu, and X. Luo, “Mining version control system for automatically generating commit comment,” in *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE Press, 2017, pp. 414–423.
- [15] A. Okutan and O. T. Yıldız, “Software defect prediction using bayesian networks,” *Empirical Software Engineering*, vol. 19, no. 1, pp. 154–181, 2014.
- [16] J. Moeyersoms, E. J. de Fortuny, K. Dejaeger, B. Baesens, and D. Martens, “Comprehensible software fault and effort prediction: A data mining approach,” *Journal of Systems and Software*, vol. 100, pp. 80–90, 2015.

- [17] S. Nelson and J. Schumann, “What makes a code review trustworthy?” in *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on.* IEEE, 2004, pp. 10–pp.
- [18] A. Edmundson, B. Holtkamp, E. Rivera, M. Finifter, A. Mettler, and D. Wagner, “An empirical study on the effectiveness of security code review,” in *International Symposium on Engineering Secure Software and Systems.* Springer, 2013, pp. 197–212.
- [19] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, “An empirical study of the impact of modern code review practices on software quality,” *Empirical Software Engineering*, vol. 21, no. 5, pp. 2146–2189, 2016.
- [20] 孙小兵, 李斌, 陈颖, 李必信, and 文万志, “软件修改影响分析研究与进展,” *电子学报*, vol. 42, no. 12, pp. 2467–2476, 2014.
- [21] S. Bouktif, Y.-G. Gueheneuc, and G. Antoniol, “Extracting change-patterns from cvs repositories,” in *Reverse Engineering, 2006. WCRE’06. 13th Working Conference on.* IEEE, 2006, pp. 221–230.
- [22] S. A. Bohner and R. Arnold, “Software change impact analysis for design evolution,” in *Proceedings of 8th International Conference on Maintenance and Reengineering.* IEEE CS Press Los Alamitos, CA, 1991, pp. 292–301.
- [23] S. Park and D.-H. Bae, “An approach to analyzing the software process change impact using process slicing and simulation,” *Journal of Systems and Software*, vol. 84, no. 4, pp. 528–543, 2011.
- [24] H. Kagdi, M. Gethers, and D. Poshyvanyk, “Integrating conceptual and logical couplings for change impact analysis in software,” *Empirical Software Engineering*, vol. 18, no. 5, pp. 933–969, 2013.
- [25] H. Abdeen, K. Bali, H. Sahraoui, and B. Dufour, “Learning dependency-based change impact predictors using independent change histories,” *Information and Software Technology*, vol. 67, pp. 220–235, 2015.

- [26] T. Apiwattanapong, A. Orso, and M. J. Harrold, “Efficient and precise dynamic impact analysis using execute-after sequences,” in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 432–441.
- [27] L. Huang and Y.-T. Song, “Precise dynamic impact analysis with dependency analysis for object-oriented programs,” in *Software Engineering Research, Management & Applications, 2007. SERA 2007. 5th ACIS International Conference on*. IEEE, 2007, pp. 374–384.
- [28] C.-Y. Lin, T.-Y. Wu, and C.-C. Huang, “Nonlinear dynamic impact analysis for installing a dry storage canister into a vertical concrete cask,” *International Journal of Pressure Vessels and Piping*, vol. 131, pp. 22–35, 2015.
- [29] H. Cai and D. Thain, “Distia: A cost-effective dynamic impact analysis for distributed programs,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 344–355.
- [30] B. Breech, M. Tegtmeyer, and L. Pollock, “Integrating influence mechanisms into impact analysis for increased precision,” in *Software Maintenance, 2006. IC-SM’06. 22nd IEEE International Conference on*. IEEE, 2006, pp. 55–65.
- [31] J. Law and G. Rothermel, “Whole program path-based dynamic impact analysis,” in *Proceedings of the 25th International Conference on Software Engineering*. IEEE Computer Society, 2003, pp. 308–318.
- [32] H. Cai and R. Santelices, “Diver: Precise dynamic impact analysis using dependence-based trace pruning,” in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 343–348.
- [33] L. Briand, Y. Labiche, and G. Soccar, “Automating impact analysis and regression test selection based on uml designs,” in *icsm*. IEEE, 2002, p. 0252.
- [34] J. Díaz, J. Pérez, J. Garbajosa, and A. L. Wolf, “Change impact analysis in product-line architectures,” in *European Conference on Software Architecture*. Springer, 2011, pp. 114–129.

- [35] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, “Using information retrieval based coupling measures for impact analysis,” *Empirical software engineering*, vol. 14, no. 1, pp. 5–32, 2009.
- [36] A. Beszedes, T. Gergely, S. Farago, T. Gyimothy, and F. Fischer, “The dynamic function coupling metric and its use in software evolution,” in *Software Maintenance and Reengineering, 2007. CSMR’07. 11th European Conference on.* IEEE, 2007, pp. 103–112.
- [37] T. Rolfsnes, S. Di Alesio, R. Behjati, L. Moonen, and D. W. Binkley, “Generalizing the analysis of evolutionary coupling for software change impact analysis,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER).* IEEE, 2016, pp. 201–212.
- [38] L. Hattori, G. dos Santos Jr, F. Cardoso, and M. Sampaio, “Mining software repositories for software change impact analysis: a case study,” in *Proceedings of the 23rd Brazilian symposium on Databases.* Sociedade Brasileira de Computação, 2008, pp. 210–223.
- [39] B. Li, X. Sun, H. Leung, and S. Zhang, “A survey of code-based change impact analysis techniques,” *Software Testing, Verification and Reliability*, vol. 23, no. 8, pp. 613–646, 2013.
- [40] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto, “Who should review my code? a file location-based code-reviewer recommendation approach for modern code review,” in *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on.* IEEE, 2015, pp. 141–150.
- [41] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, “Investigating code review quality: Do people and participation matter?” in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on.* IEEE, 2015, pp. 111–120.

- [42] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, “Investigating technical and non-technical factors influencing modern code review,” *Empirical Software Engineering*, vol. 21, no. 3, pp. 932–959, 2016.
- [43] V. Murali, S. Chaudhuri, and C. Jermaine, “Bayesian specification learning for finding api usage errors,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 151–162.
- [44] C. Mills, “Automating traceability link recovery through classification,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 1068–1070.
- [45] M. Rath, J. Rendall, J. L. Guo, J. Cleland-Huang, and P. Mäder, “Traceability in the wild: automatically augmenting incomplete trace links,” in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 834–845.
- [46] S. Karim, H. L. H. S. Warnars, F. L. Gaol, E. Abdurachman, B. Soewito *et al.*, “Software metrics for fault prediction using machine learning approaches: A literature review with promise repository dataset,” in *Cybernetics and Computational Intelligence (CyberneticsCom), 2017 IEEE International Conference on*. IEEE, 2017, pp. 19–23.
- [47] K. Shimonaka, S. Sumi, Y. Higo, and S. Kusumoto, “Identifying auto-generated code by using machine learning techniques,” in *Empirical Software Engineering in Practice (IWESEP), 2016 7th International Workshop on*. IEEE, 2016, pp. 18–23.
- [48] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, “Statistical learning approach for mining api usage mappings for code migration,” in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 457–468.

- [49] M. Linares-Vásquez, C. McMillan, D. Poshyvanyk, and M. Grechanik, “On using machine learning to automatically classify software applications into domain categories,” *Empirical Software Engineering*, vol. 19, no. 3, pp. 582–618, 2014.
- [50] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, “Predicting source code changes by mining change history,” *IEEE transactions on Software Engineering*, vol. 30, no. 9, pp. 574–586, 2004.
- [51] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, “Mining version histories to guide software changes,” *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.
- [52] N. Ajienka, A. Capiluppi, and S. Counsell, “An empirical study on the interplay between semantic coupling and co-change of software classes,” *Empirical Software Engineering*, vol. 23, no. 3, pp. 1791–1825, 2018.
- [53] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, and G. Antoniol, “Detecting asynchrony and dephase change patterns by mining software repositories,” *Journal of Software: Evolution and Process*, vol. 26, no. 1, pp. 77–106, 2014.
- [54] S. Vaucher, H. Sahraoui, and J. Vaucher, “Discovering new change patterns in object-oriented systems,” in *Reverse Engineering, 2008. WCRE’08. 15th Working Conference on.* IEEE, 2008, pp. 37–41.
- [55] B. Fluri, E. Giger, and H. C. Gall, “Discovering patterns of change types,” in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering.* IEEE Computer Society, 2008, pp. 463–466.
- [56] A. Michail, “Data mining library reuse patterns in user-selected applications,” in *ase.* IEEE, 1999, p. 24.
- [57] Y. Lin and D. Dig, “Check-then-act misuse of java concurrent collections,” in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation.* IEEE, 2013, pp. 164–173.

- [58] R. Holmes, R. J. Walker, and G. C. Murphy, “Approximate structural context matching: An approach to recommend relevant examples,” *IEEE Transactions on Software Engineering*, no. 12, pp. 952–970, 2006.
- [59] M. Bruch, M. Monperrus, and M. Mezini, “Learning from examples to improve code completion systems,” in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 213–222.
- [60] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen, “Mining software repositories to study co-evolution of production & test code,” in *Software Testing, Verification, and Validation, 2008 1st International Conference on*. IEEE, 2008, pp. 220–229.

致谢