

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 提交库构建	13
2.3 相似提交的检索	17
2.4 关键类判定方法	21
2.5 影响分析辅助方法	22
2.6 实验结果与分析	27
2.7 本章小结	31
第 3 章 基于可判别特征的代码修改周期预测	33
3.1 问题描述及方法总览	33
3.2 可判别特征的提取	34
3.3 机器学习算法选择及模型评估方法	40
3.4 代码修改周期预测实验设置与评估	42
3.5 本章小结	47
第 4 章 代码修改分析与注释检查系统	48
4.1 代码修改分析模块	48
4.2 注释检查模块	51
4.3 基础服务模块	53
4.4 本章小结	53

第 5 章 总结与展望	59
5.1 工作总结	59
5.2 研究展望	60
参考文献	62
致谢	71

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

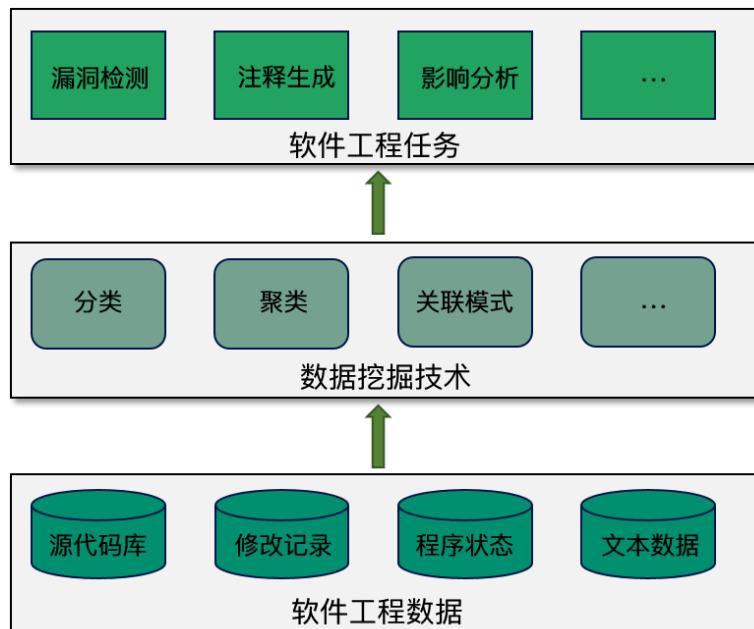


图 1-1 软件工程数据挖掘流程

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>

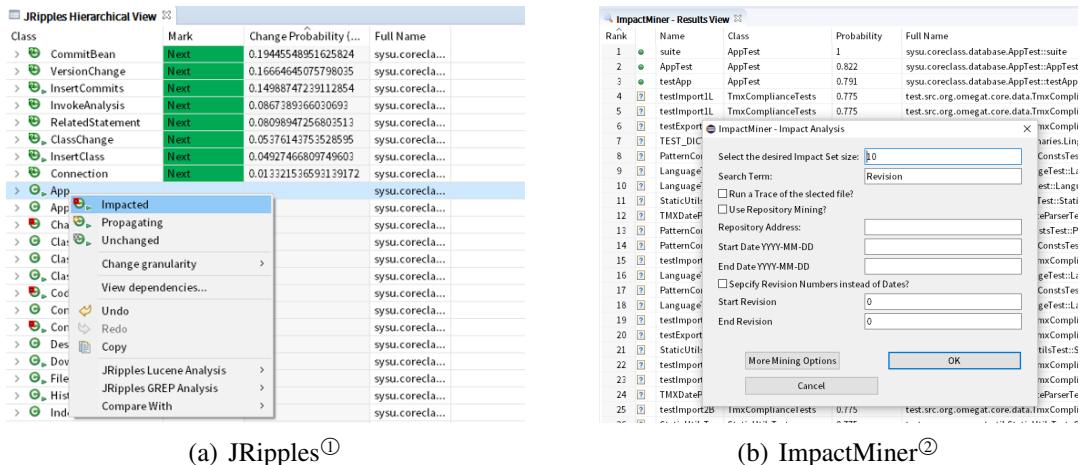


图 1-4 修改影响分析工具

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，构建历史提交库。第二步，使用第一步中预处理后文本信息构建文本语料库，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，使用词嵌入方法Word2vec训练得到词嵌入模型；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；利用词嵌入模型计算当前修改与历史提交的向量相似度，

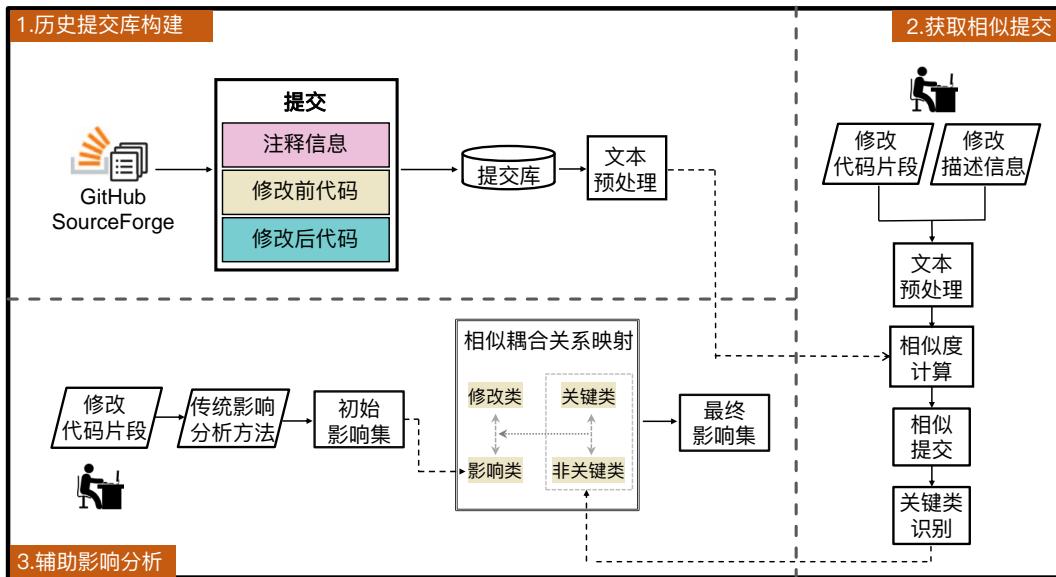


图 2-1 基于历史修改模式的影响分析辅助方法总览

得到相似修改提交列表。第三步，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 提交库构建

本文中用于构建提交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库。

2.2.1 提交数据筛选

本文用于构建提交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提交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，会在后续详相似提交检索对描述信息的对比产生负面影响；（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，一个提交中存在多个不相关的修改内容。

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。

2.2.2 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“*imformation*”。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，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。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，



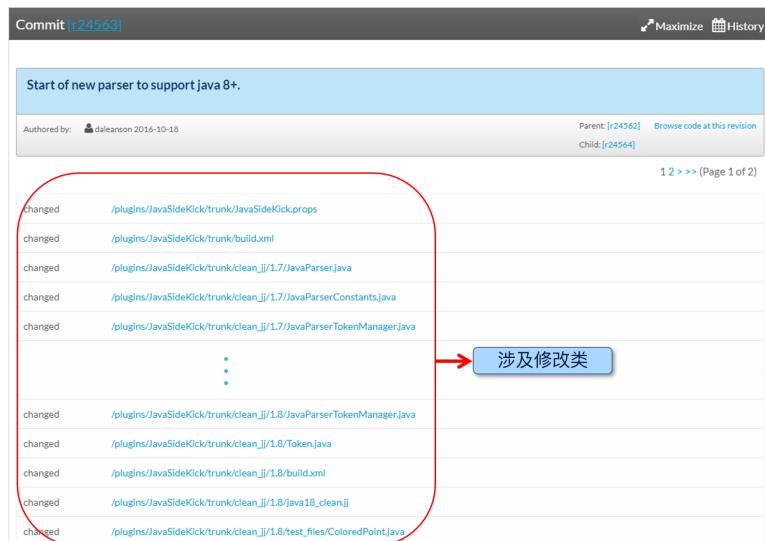
(a) 提交注释信息过短



(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 存在问题的提交图例

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，还需要对当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。

2.2.3 提交语料库构建

本文提出的方法需要从本地提交库中检索与当前修改任务最相似的代码提交，其中关键点是计算当前修改代码、修改描述与提交中修改代码、提交注释之间的相似度。提交中的注释文本和代码文本经过预处理后可以融合成一条代表提交的语料信息，我们通过所有提交的语料信息构建提交语料库。其中，值得注意的是，对于代码文本（无论是提交中代码，还是当前修改任务中代码）首先需要识别其中涉及修改的代码段。另外，一个提交中通常涉及多个类的修改，需要识别其中核心修改的类，作为当前修改类的等价类。

本文使用*ChangeDistiller*方法[61] 中修改前后两个版本的代码文本中提取涉及修改的代码段。*ChangeDistiller*方法通过对比两个版本代码对应的抽象语法树之间的差异来获取涉及修改的代码段。由于提交中存在多种修改任务，*ChangeDistiller*方法还能识别提交中不同的代码修改类型，例如：参数名变更（“*Parameter Renaming*”），参数增加/删除（“*Parameter Insert/Delete*”），方法名变更（“*Method Renaming*”），语句增加/删除（“*Statement Insert/Delete*”）等。另外，我们使用关键类判定方法识别提交中核心修改的类。关键类判定方法将在后文介绍。

对于提交中注释文本、代码修改前片段及代码修改后片段，我们分两方面

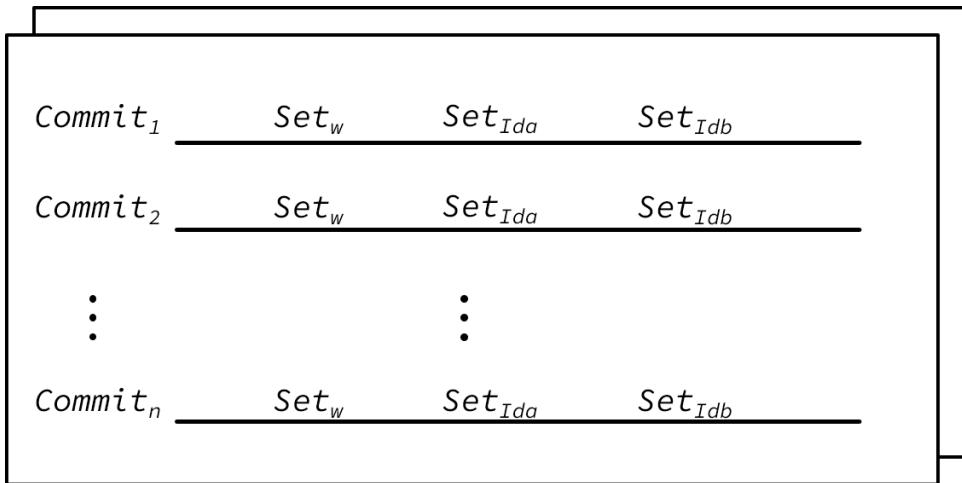


图2-3 提交文本语料库构建方式图示

处理，首先将三个文本储存于提交库中用于后续相似提交检索，此外，将提交库中所有提交的文本语料融合成语料库，用于训练词嵌入模型。本文语料库的组织形式如下：

对于注释文本中每个单词 w_i ，从修改前代码文本中随机选取四个标识符 Id_{a1} 、 Id_{a2} 、 Id_{a3} 、 Id_{a4} ，组合成 Id_{a1} 、 Id_{a2} 、 w_i 、 Id_{a3} 、 Id_{a4} ，记为 Com_1 ；从修改后代码文本中随机选取四个标识符 Id_{b1} 、 Id_{b2} 、 Id_{b3} 、 Id_{b4} ，组合成 Id_{b1} 、 Id_{b2} 、 w_i 、 Id_{b3} 、 Id_{b4} ，记为 Com_2 ；从而，针对注释文本中每个单词得到组合语料 $Com_{w_i} = \{Com_1, Com_2\}$ ，所有 Com_{w_i} 组合得到注释文本对应语料 Set_w 。

对于修改前代码中每个标识符 Id_{ai} ，则随机从注释文本中选取四个单词 w_1 、 w_2 、 w_3 、 w_4 ，与 Id_{ai} 组合成 w_1 、 w_2 、 Id_{ai} 、 w_3 、 w_4 ，记为 Com_{Id_a} ，所有的 Com_{Id_a} 组合得到修改前代码对应语料 Set_{Id_a} ；类似地，对于修改后代码中每个标识符 Id_{bi} ，随机从注释文本中选取四个单词 w_1 、 w_2 、 w_3 、 w_4 ，与 Id_{bi} 组合成 w_1 、 w_2 、 Id_{bi} 、 w_3 、 w_4 ，记为 Com_{Id_b} ，所有的 Com_{Id_b} 组合得到修改前代码对应语料 Set_{Id_b} 。

最后，将每个提交对应的 Set_w 、 Set_{Id_a} 、 Set_{Id_b} 组合成一条语料，所有提交对应的语料信息组合得到完整的语料库（如图2-3所示）。

2.3 相似提交的检索

本节详细介绍了从本地提交库中检索相似提交的过程。我们通过提交语料库训练词向量模型，再根据词向量模型计算提交与当前修改对象之间的相似度，

筛选出相似度最高的前20个提交用于后续影响分析结果优化。提交与当前对象的相似度结合了修改代码相似度以及修改文本描述的相似度。

2.3.1 词向量模型训练

当前，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^①的语义相似度计算方法。WordNet由庞大的词汇数据库构成，通过同义词集形成词汇网络。其中，同义词通过概念-语义和词汇关系相互关联。两个单词间的语义相似度可以通过计算他们在WordNet构成的单词网络中的所在位置的距离衡量。另一种是基于词嵌入技术（Word Embedding）的语义相似度计算方法。词嵌入技术使用多层神经网络将单词投射到语义空间中，从而可以确定词与词之间的语义距离或语义相似度。其核心思想是通过嵌入一个线性的投影矩阵，将稀疏的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）映射为一个稠密的联系向量，并通过一个语言模型的任务去学习向量的权重。

本文正是采用基于词嵌入技术的Word2vec方法[62]训练词向量模型。Word2vec的基本思想是通过训练将每个词映射为K维向量（K是人为设定的超参数），再根据词与词之间的向量距离来确定它们的语义相似度。其模型由三层神经网络构成，输入层-隐藏层-输出层，通过三层神经网络对语言模型进行建模，从而得到单词在向量空间上的表示。与潜在语义分析（Latent Semantic Index）、潜在狄利克雷分配（Latent Dirchlet Allocation）相比，Word2vec充分利用了上下文信息，使得语义信息更加准确。Word2vec的训练模型主要分为两种，CBOW模型和Skip-Gram模型（如图2-4所示）。这两种模型主要的区别在于词向量模型训练过程中，CBOW模型根据上下文信息预测目标单词的概率分布，而Skip-Gram模型则是通过当前单词预测其上下文的概率[63, 64]。

本文采用的词向量模型为Skip-Gram模型，模型的训练目标是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 $2k + 1$ 的上下文滑动窗口中的中心词和中心词的上下文(本文中取 $k = 2$ ，即滑动窗口大小为5)，n代表语句的长度。式

^① <https://wordnet.princeton.edu/>

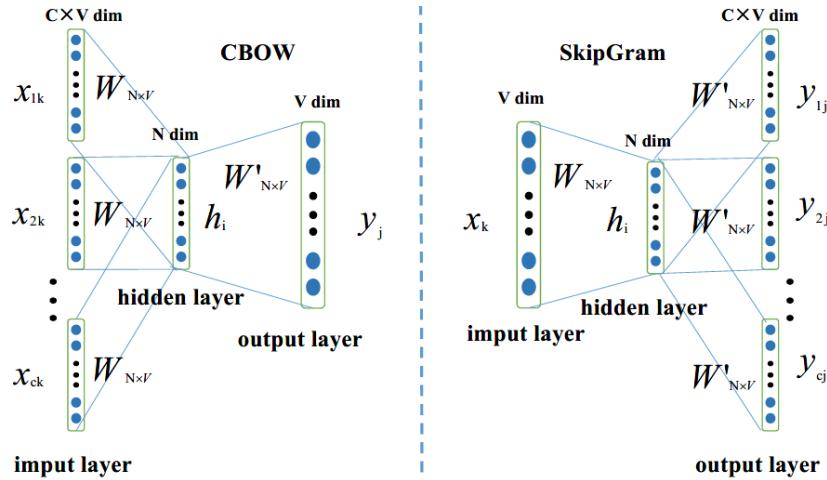


图 2-4 CBOW与Skip-Gram模型区别

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:

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

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。

2.3.2 获取相似提交列表

通过以上方法得到词向量模型后, 我们可以得到提交语料中每个单词的向量表示。本文根据向量的余弦距离度量两个文本之间的相似度, 再筛选出相似度最高的前20个提交构成相似提交列表。文本相似度的计算方法如下:

对于文本 C 、文本 S 中任意单词 w_c 和 w_s , 其中 $w_c \in C$, $w_s \in S$, w_c 和 w_s 相似度为:

$$sim(w_c, w_s) = \cos(\mathbf{w}_c, \mathbf{w}_s) = \frac{\mathbf{w}_c^T \mathbf{w}_s}{\|\mathbf{w}_c\| \|\mathbf{w}_s\|} \quad (2.3)$$

对于单词 w_c 与文本 S 的语义相似度, 则取单词 w_c 与文本 S 中所有单词之间相似度的最高值, 公式如下:

$$sim(w_c, S) = \max_{w_s \in S} sim(w_c, w_s) \quad (2.4)$$

由于在文本相似度计算中TF-IDF值作为单词权重的方法会造成近似相交性[63]。本文采用文本C中单词与文本S相似度的平均值作为文本C到文本S的相似度。另外，我们忽略与文本相似度为0的单词，作如下集合定义：

$$Set(C \rightarrow S) = \{w_c \in C | sim(w_c, S) \neq 0\} \quad (2.5)$$

文本C到文本S的相似度为：

$$sim(C \rightarrow S) = \frac{\sum_{w_c \in Set(C \rightarrow S)} sim(w_c, S)}{|Set(C \rightarrow S)|} \quad (2.6)$$

同理，文本S到文本C的相似度为：

$$sim(S \rightarrow C) = \frac{\sum_{w_s \in Set(S \rightarrow C)} sim(w_s, C)}{|Set(S \rightarrow C)|} \quad (2.7)$$

由此，我们可以得到文本C与文本S的相似度：

$$sim(C, S) = \frac{sim(C \rightarrow S) + sim(S \rightarrow C)}{2} \quad (2.8)$$

本文中提交与影响分析对象的相似度有三部分度量：（1）提交注释文本与影响分析对象修改目标的相似度*CommentSim*；（2）提交中旧版本代码修改片段与影响分析对象修改前代码片段的相似度*OldCodeSim*；（3）提交中新版本代码修改片段与影响分析对象修改后代码片段的相似度*NewCodeSim*。三者赋予不同的权重得到提交的综合相似度*CommitSimi*，公式如下：

$$CommitSimi = \alpha \cdot CommentSimi + \beta \cdot OldCodeSimi + \gamma \cdot NewCodeSimi \quad (2.9)$$

在实验测试中得到最优权重组合为: $\alpha = 0.4$ 、 $\beta = 0.3$ 、 $\gamma = 0.3$ 。

2.4 关键类判定方法

本文提出的影响分析辅助方法中一个关键步骤是从相似提交中找出影响分析对象的等价类, 使用等价类的修改模式对初始影响分析结果进行优化。本文通过关键类判定方法识别提交中核心修改的类, 将关键类作为影响分析对象的等价类。我们从代码耦合、代码修改以及提交类型三个维度提取特征(如表2-1所示), 训练机器学习模型, 通过机器学习模型判别提交中的关键类。本节主要介绍三个维度的特征提取方法。

代码耦合特征是一次代码提交中, 多个类修改之间的代码耦合关系。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; 出度指的是提交中被该类引用的其他类的个数。同时, 我们还考虑了提交中不同类之间的出度与入度的关系。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, 更可能成为提交中的关键类。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影响比较大, 这个类有比较高的概率是非关键类。一个类的入度和出度的不同组合也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修改的影响都比较大。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, 这个类更可能是非关键类。

对于一个提交, 类的代码修改量往往对应着这个类在提交中的重要程度, 关键类在一次提交中往往是被主要修改的类。从修改代码量维度衡量, 关键类修改的代码量通常要比非关键类多。类修改的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。另外, 类的修改类型可以分为新增, 修改和删除三种。我们定义的代码修改特征如表。

在一次提交中, 代码修改的目标有几种类型: (1) 添加新功能(前向工

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。

表 2-1 关键类判别特征

类型	编号	描述
代码耦合特征	01	入度
	02	入度/提交中类最大入度
	03	入度/提交中类平均入度
	04	出度
	05	出度/提交中类最大出度
	06	出度/提交中类平均出度
	07	入度是否为零
	08	出度是否为零
代码修改特征	09	代码修改行数
	10	代码修改行数/提交中类最大代码修改行数
	11	代码修改行数/提交中类平均代码修改行数
	12	方法修改数
	13	方法修改数/类中方法总数
	14	方法修改数/提交中最大方法修改数
	15	方法修改数/提交中平均方法修改数
	16	修改类型
提交类型特征	17	是否为前向工程
	18	是否为逆向工程
	19	是否为纠错工程
	20	是否非代码类型修改
	21	提交中类的个数

2.5 影响分析辅助方法

本节详细介绍基于相似提交对传统影响分析的修改影响集进行优化的过程。我们先使用传统影响分析方法得到修改类的初始影响集, 初始影响集中的类称

之为初始影响类。从提交库中检索得到前20个最相似的提交，提取提交中关键类与其他类的结构耦合关系以及修改类与初始影响类的结构耦合关系，计算结构耦合关系的相似度，对相似度高于指定阈值的初始影响类，调整其在初始影响集的位置，得到最终影响集。

2.5.1 软件实体间结构耦合关系提取

我们提出的基于历史修改模式的影响分析辅助方法中，通过类与类之间结构耦合关系的相似性，将历史修改模式映射回当前修改类中。当相似提交中关键类A和其他类B的结构耦合关系与当前修改类和初始影响类C的耦合关系的相似度很高时，我们认为修改类更可能将修改影响传播给初始影响类C。我们分析的软件实体包括类、方法和属性。对于两个类A和B（或者类A，接口B），我们从类层次、方法层次以及属性层次总结了A、B的18种耦合关系（如表2-2所示）：

- 1) A类的类层次。主要有2种耦合类型：A类继承B类、A类实现B接口。
- 2) A类的方法层次（但没有使用B类定义局部变量或调用B类中的静态变量、静态方法、构造方法）。主要有5种耦合类型：强制类型转换、类型检查(instanceOf)、B作为方法返回值类型、类实例、方法抛异常（B为异常类）。
- 3) A类的属性层次（成员属性、方法局部属性、方法参数），即，以B类定义A类的属性attr，attr有多种不同的使用方式，我们考虑以下三种：
 - 3.1) 在A类中（成员属性、方法局部属性、方法参数）直接使用attr；
 - 3.2) 在A类中（成员属性、方法局部属性、方法参数）调用attr（attr为B的实例）的某个属性；
 - 3.3) 在A类中（成员属性、方法局部属性、方法参数）调用attr（attr为B的实例）的某个方法；
- 4) 其他情况：在A类的方法（也可能在A类的属性、静态代码块、非静态代码块，根据调研这三种情况较少见）调用B的静态方法或B的构造方法，共两种耦合关系。

两个类之间的18种耦合关系我们通过一个1个18维的特征向量表示，每个维度对应表2-2的编码。如果两个类之间满足某种耦合关系，该维度表示为1，否

则为0。

表2-2 类与类（接口）耦合类型表

层次	编码	耦合类型
类层次	01	A类继承B类
	02	A类实现B接口
方法层次	03	强制类型转换
	04	类型检查instanceOf
	05	B类作为A方法返回值类型
	06	类实例b.Class
	07	方法抛异常（B为异常类）
属性层次	08	直接调用attr 通过attr调用其成员属性（attr为B类的实例） 通过attr调用其成员方法（attr为B类的实例）
	09	
	10	
	11	直接调用attr 通过attr调用其成员属性（attr为B类的实例） 通过attr调用其成员方法（attr为B类的实例）
	12	
	13	
	14	直接调用attr 通过attr调用其成员属性（attr为B类的实例） 通过attr调用其成员方法（attr为B类的实例）
	15	
	16	
其他情况	17	A类调用B类的静态方法
	18	A类调用B类的构造方法

2.5.2 初始影响集优化

通过传统影响分析方法获得初始影响集后，我们对影响集列表中的类赋予初始分数如表2-3所示。影响集中类的排序代表着其受影响的概率，因此，我们记倒数第一个类为0分，逆序为排在前面的类依次增加20分。

对于初始影响集中的每个类，首先，提取其与修改类之间的结构耦合关系，记耦合关系为 V_i （18维向量）；然后，提取每个相似提交中关键类与其他类的耦合关系，记为 V_j ；最后，计算 V_i 与所有 V_j 相似度，若 V_i 与提交中有 n 个 V_j 相似度大于指定阈值 η （由实验测试得出），则 V_i 所对应的初始影响类分数增加 $n * 20$ 分，调整其在影响集中的位置。

本文通过Jaccard相似系数度量两组耦合关系向量之间的相似性，Jaccard相似系数相对于其他相似性度量方法，更适合处理非对称二元变量。在两组结构

表 2-3 初始影响集分数赋值示例

排序	类名	影响概率	初始分数
1	<i>Buffer.java</i>	0.7374193	240
2	<i>EditServer.java</i>	0.6746630	220
3	<i>Macros.java</i>	0.6488870	200
4	<i>EditPlugin.java</i>	0.6138285	180
5	<i>GUIUtilities.java</i>	0.5360856	160
6	<i>BeanShell.java</i>	0.5250749	140
7	<i>Abbrevs.java</i>	0.4440035	120
8	<i>BufferIORequest.java</i>	0.3989099	100
9	<i>FileVFS.java</i>	0.3953681	80
10	<i>Mode.java</i>	0.2678907	60
11	<i>View.java</i>	0.1140896	40
12	<i>EditPane.java</i>	0.0990951	20
13	<i>HistoryModel.java</i>	0.0719621	0

耦合关系中，某一维度上的耦合关系编码正匹配（两者都取值为1）比负匹配（两者都取值为0）更有意义，在相似性度量上，负匹配的数量被认为是不重要的，*Jaccard*相似系数可以忽略负匹配的影响。初始影响集优化公式如下：

$$JaccardSimi(V_i, V_j) = \frac{V_i \cap V_j}{V_i \cup V_j} \quad (2.10)$$

$$Score_i = \begin{cases} Score_i + 20, & JaccardSimi(V_i, V_j) > \eta \\ Score_i, & \text{else} \end{cases} \quad (2.11)$$

2.5.3 基于历史修改模式的影响分析算法

算法2-1是本文提出的影响分析辅助方法的算法整体流程，本节主要阐述算法的实现流程及细节。算法以当前修改目标的自然语言描述*describe*，代码修改片段*codefragList*作为输入；输出目标为初始影响集优化后得到的最终影响集*finImpactedSet*。

算法3-15行为相似提交的检索过程。通过2.3节中提交相似度计算方法，对比提交库中的所有提交，得到一个大小为20的相似提交列表*SimiCommits*。在检

算法2-1: 影响分析算法

```
1: Input: describe: 修改目标, codefragList: 修改代码片段;  
2: Output: finImpactedSet: 最终影响集;  
3: SimiCommits = {}  
4: foreach commit ∈ commitCorpus  
5:   simi = getTextSimi(commit, describe, codefragList)  
6:   if SimiCommits.size() < 20 do  
7:     SimiCommits.add(commit)  
8:   end if  
9:   Sorted(SimiCommits)  
10:  if SimiCommits.size() == 20 && simi > SimiCommits.get(SimiCommits.size()-1).simi do  
11:    SimiCommits.add(commit)  
12:    Sorted(SimiCommits)  
13:    SimiCommits.remove(SimiCommits.size()-1)  
14:  end if  
15: end foreach  
16: initImpactedSet = getInitImpactedSet(describe, codefragList)  
17: foreach impactClass ∈ initImpactedSet do  
18:   impactClassCoupleVec = get CoupleVec(changeClass, impactClass)  
19:   foreach commit ∈ SimiCommits do  
20:     coreclass = getCoreClass(commit)  
21:     foreach otherclass ∈ commit do  
22:       otherClassCoupleVec = get CoupleVec(coreclass, otherclass)  
23:       JaccardSimi = getJaccard(impactClassCoupleVec, otherClassCoupleVec)  
24:       if JaccardSimi > 0.8 do  
25:         impactClass.Score += 20  
26:       end if  
27:     end foreach  
28:   end foreach  
29: end foreach  
30: finImpactedSet = sorted(initImpactedSet)  
31: return finImpactedSet
```

索提交库过程中，若相似提交列表中提交数量低于20，则直接加入当前检索的提交；若提交列表中提交数量为20，则对比提交列表中最后一个提交（相似度最低的提交）的相似系数与当前提交的相似系数的大小，如果当前提交的相似系数更大，则更新提交列表。

算法16-30行为初始影响集*initImpactedSet*的优化过程。*initImpactedSet*由传统影响分析方法得到，并对影响集中的类根据受影响概率不同赋予不同的初始分数*Score*。对于*initImpactedSet*中一个受影响的类*impactClass*，提取其与修改类的结构耦合关系*impactClassCoupleVec*；然后遍历相似提交列表*SimiCommits*中所有提交*commit*，识别提交中的关键类*coreclass*，对于提交中的所有非关键类*otherclass*，提取其与关键类的结构耦合关系*otherClassCoupleVec*，再计算*impactClassCoupleVec*与*otherClassCoupleVec*的Jaccard相似系数，若*JaccardSimi*大于指定阈值（实验测试最优阈值为0.8），则当前受影响类*impactClass*对应的分数*Score*增加20；最后对初始影响集根据分数进行重排序得到最终影响集*finImpactedSet*。

2.6 实验结果与分析

在本文中将介绍影响分析辅助方法的实验设计与评估。首先介绍实验数据的选取过程，然后介绍实验的评估标注，最后给出实验结果。

2.6.1 数据集收集

我们从开源项目中选取8个项目进行影响分析实验。这些项目均保持着长期维护，有大量优质提交信息，并且是在软件工程领域常用的开源项目。这些项目分别是FreeCol^①，HSQLDB^②，HtmlUnit^③，JAMWiki^④，jEdit^⑤，jHotDraw^⑥，Makagiga^⑦，OmegaT^⑧。这些开源项目代表了不同的应用领域和开发环境。

同时，为了保证实验的可靠性，需要使用本文2.2.1节中提出的提交数据优化方法对实验数据集进行优化，去除只涉及一个修改类的提交（该类提交不包含影响集）以及涉及20个修改类的提交（该类提交通常由多个普通提交组合而成）。项目的详细信息如表2-4所示。

^① <http://www.jamwiki.org/>

^② <http://hsqldb.org/>

^③ <http://htmlunit.sourceforge.net/>

^④ <http://www.jamwiki.org/>

^⑤ <http://www.jedit.org/>

^⑥ <http://www.jhotdraw.org/>

^⑦ <https://makagiga.sourceforge.io/>

^⑧ <https://omegat.org/>

表 2-4 实验项目数据信息

项目名	版本跨度	提交个数	项目名	版本跨度	提交个数
FreeCol	0.7-1.0	1150	HSQLDB	2.20-2.33	637
HtmlUnit	2.0-2.21	737	JAMWiki	0.82-1.3	499
jEdit	4.0-4.5	1601	JHotDraw	7.1-7.5	378
Makagiga	3.82-4.12	2072	OmegaT	2.3-3.54	627

2.6.2 实验评估准则

影响分析研究中广泛使用召回率和精确率作为结果的度量指标，这两个指标能评估实验产生的预估影响集与真实修改产生的实际影响集之间的差异。给定软件实体 e_s （本文影响分析实体为类），用 E_{imp} 表示影响分析结果中软件实体的集合（预估影响集），用 R_{imp} 表示实际修改中被影响的软件实体的集合（实际影响集）。则实验结果的召回率由公式2.12定义，精确率由公式2.13定义。此外，为了更细致的评估实验结果，我们对预估影响集按不同截断点进行划分，分别评估实验影响集前5、前10、前20、前50的召回率和精确率，如公式2.14和公式2.15定义。

$$Recall = \frac{1}{n} \sum_{i=1}^n \frac{|E_{imp} \cap R_{imp}|}{|R_{imp}|} \times 100\% \quad (2.12)$$

$$Precision = \frac{1}{n} \sum_{i=1}^n \frac{|E_{imp} \cap R_{imp}|}{|E_{imp}|} \times 100\% \quad (2.13)$$

$$Recall(cutpoint) = \frac{1}{n} \sum_{i=1}^n \frac{|E_{imp}(cutpoint) \cap R_{imp}|}{|R_{imp}|} \times 100\% \quad (2.14)$$

$$Precision(cutpoint) = \frac{1}{n} \sum_{i=1}^n \frac{|E_{imp}(cutpoint) \cap R_{imp}|}{cutpoint} \times 100\% \quad (2.15)$$

为了验证本文提出的影响分析辅助方法的有效性，我们选取两个影响分析工具*JRipples*和*ImpactMiner*作为传统影响分析方法，用于产生初始影响集。*JRipples*工具是Eclipse官方插件，该插件是Java代码影响分析领域中被广泛使用的工具，也是实际开发中开发人员常用的影响分析工具。而*ImpactMiner*则是可用工具中影响分析效果较好的一款工具，另外，*ImpactMiner*工具结合了多种影响分析中的研究方法，包括：信息检索，软件历史库挖掘以及动态影响

分析。

本文实验中的验证数据为开源项目中提交数据，无法直观判断其修改集和影响集。因此，我们通过关键类判定方法识别提交中的关键类，将关键类作为实验的修改集，提交中所有非关键类作为影响集。

在实验中，我们主要关注两个方面的问题：第一，我们关注的是本文提出的影响分析辅助方法，是否能提高传统影响分析工具的召回率和准确率？第二，在历史修改模式映射中，结构耦合关系的相似程度是否影响召回率和准确率？

2.6.3 实验结果分析

研究问题1：我们的方法是否能提高传统影响分析的效果？

我们分别基于*JRipples*工具和*ImpactMiner*工具在8个项目上评估了影响分析辅助方法效果，实验结果如表2-5和表2-6所示。实验结果显示，我们的方法在8个项目上对两个工具的初始影响集都起到了辅助效果，证实了本文提交出方法都有效性。观察可以发现，我们的方法对*ImpactMiner*的辅助效果更好，这是由于我们在实验中设置初始影响集的大小为50，而*ImpactMiner*的初始影响集中包含更多实际影响类，调整空间更大。从表2-5中可以发现，在JAMWiki和OmegaT两个项目的实验结果中前30和前50的召回率和精确率是相同的，这是由于我们*JRipples*在这两个项目上影响分析产生的初始影响集大小小于30，导致我们取前30和前50去评估实验结果是相同的。同时，我们可以发现，当传统影响分析方法前50召回率或精确率更高时，我们的方法的辅助效果更加明显。这是因为初始影响集中包含更多的实际影响类，在影响集重排序时，我们的方法有更大几率将实际影响类的排序位置往前移动。

综上所述，本文提出的基于历史修改模式的影响分析辅助方法在多个项目上能提升传统影响分析工具的效果，验证了本文方法的有效性。

研究问题2：结构耦合关系相似程度是否影响召回率和精确率？

对于第二个研究问题，我们以*JRipples*为传统影响分析工具在3个项目上测试不同结构耦合关系相似程度对召回率和精确率的影响。如本文2.5.2节所述，相似度阈值为 η ，当提交中关键类和一个非关键类的耦合关系与修改类与一个初始影响集中影响类的耦合关系的相似度大于阈值，认为修改类的修改影响会传播给该影响类，并增加该影响类的分数。实验测试不同 η 下影响分析的召回率和

表2-5 基于JRipples的影响分析辅助结果

项目名	方法	召回率 (%)					精确率 (%)				
		5	10	20	30	50	5	10	20	30	50
FreeCol	JRipples	13.68	20.52	25.50	28.15	28.72	12.65	9.53	6.01	4.37	2.68
	Our Method	14.65	20.42	25.49	28.72	28.72	13.65	9.70	28.72	28.72	2.68
HSQLDB	JRipples	16.43	24.76	42.03	55.67	61.05	22.56	18.33	15.13	13.04	8.93
	Our Method	18.10	30.53	47.12	55.67	61.05	24.56	21.14	16.18	13.04	8.93
HtmlUnit	JRipples	15.17	18.93	25.19	26.76	28.63	11.55	7.25	4.79	3.48	2.28
	Our Method	17.00	19.89	25.91	27.30	28.63	13.14	8.74	4.82	3.52	2.28
JAMWiki	JRipples	24.08	34.12	38.60	40.07	40.16	16.30	12.35	7.35	5.15	3.11
	Our Method	32.27	9.70	38.90	40.07	40.16	23.19	13.69	13.69	13.69	3.11
jEdit	JRipples	23.59	34.18	43.77	47.28	50.10	16.15	11.15	6.90	5.19	3.31
	Our Method	29.41	37.23	44.83	48.34	50.10	18.07	12.12	7.02	5.32	3.11
JHotDraw	JRipples	12.82	16.86	17.29	18.34	18.82	11.07	8.39	4.38	3.21	2.04
	Our Method	15.10	18.45	17.39	18.34	18.82	13.77	13.77	4.44	3.21	2.04
Makagiga	JRipples	21.78	30.67	39.54	46.34	51.74	24.04	17.41	12.13	9.78	6.99
	Our Method	23.22	31.32	40.09	49.83	51.48	25.74	17.93	12.30	9.91	6.99
OmegaT	JRipples	39.78	43.93	44.54	44.54	44.54	22.04	12.04	6.02	6.02	6.02
	Our Method	41.41	44.03	44.54	44.54	44.54	23.65	13.04	6.02	6.02	6.02

表2-6 基于ImpactMiner的影响分析辅助结果

项目名	方法	召回率 (%)					精确率 (%)				
		5	10	20	30	50	5	10	20	30	50
FreeCol	JRipples	18.22	24.78	31.23	34.78	38.39	18.15	14.63	12.92	9.63	5.31
	Our Method	22.47	28.68	33.21	35.98	38.39	22.12	17.35	14.21	10.39	5.31
HSQLDB	JRipples	34.33	39.03	47.99	58.33	63.87	30.13	24.11	21.69	17.56	14.02
	Our Method	41.09	45.42	51.08	60.77	63.87	33.77	27.65	23.00	18.14	14.02
HtmlUnit	JRipples	18.33	22.43	28.06	29.97	31.70	15.48	11.98	9.97	7.53	5.89
	Our Method	20.17	24.08	29.61	31.15	31.70	17.44	13.70	11.31	9.65	5.89
JAMWiki	JRipples	33.80	38.21	41.26	45.04	49.19	19.09	16.38	12.82	9.95	7.38
	Our Method	39.10	42.44	45.37	47.39	49.16	23.41	18.64	14.63	11.85	7.38
jEdit	JRipples	34.39	47.13	54.22	57.33	62.12	15.13	21.32	18.54	15.23	11.33
	Our Method	41.44	53.33	58.83	60.34	62.12	30.11	28.49	23.12	19.83	11.33
JHotDraw	JRipples	16.72	21.02	24.26	27.11	29.30	13.07	11.55	9.87	8.05	5.15
	Our Method	18.25	24.54	27.24	28.12	29.30	14.25	12.54	11.06	8.49	5.15
Makagiga	JRipples	28.31	35.76	40.01	48.21	53.87	29.33	23.54	17.65	13.33	10.94
	Our Method	33.64	39.09	44.84	51.59	53.87	33.04	27.81	22.02	16.52	10.94
OmegaT	JRipples	44.34	50.43	53.66	56.71	61.63	26.54	22.76	19.42	16.21	13.11
	Our Method	51.39	54.93	58.01	59.82	61.63	29.45	24.35	21.10	18.61	13.11

精确率。实验结果如表2-7所示，我们可以发现当相似度阈值 η 从0.3, 0.5, 0.8一次增加时，影响分析的召回率和精确率在不断提升，但 η 从0.8增加到0.9时，影响分析的召回率和精确率是下降的。同时我们可以发现， η 取值为0.3时，影响分

表2-7 不同相似阈值 η 下实验结果

项目名	方法	η	召回率 (%)					精确率 (%)			
			5	10	20	30	50	5	10	20	30
jEdit	Our method	JRipples	23.59	34.18	43.77	47.28	50.10	16.15	11.15	6.90	5.19
		0.3	17.23	27.32	35.11	42.65	50.10	13.17	9.88	6.34	5.19
		0.5	23.62	33.75	42.68	46.44	50.10	16.21	11.66	6.90	5.24
		0.8	29.41	37.23	44.83	48.34	50.10	18.07	12.12	7.02	5.32
		0.9	26.67	35.54	43.12	47.19	50.10	16.46	12.08	6.95	5.32
JHotDraw	Our method	JRipples	16.72	21.02	24.26	27.11	29.30	13.07	11.55	9.87	8.05
		0.3	13.31	16.76	20.01	25.59	29.30	9.81	7.65	6.02	5.52
		0.5	16.76	20.87	25.35	28.81	29.30	12.84	11.41	9.25	8.13
		0.8	18.25	24.54	27.24	28.12	29.30	14.25	12.54	11.06	8.49
		0.9	17.09	23.84	25.59	27.36	29.30	13.57	12.38	10.88	8.21
HSQLDB	Our method	JRipples	12.82	16.86	17.29	18.34	18.82	11.07	8.39	4.38	3.21
		0.3	8.54	12.10	15.89	16.34	18.82	8.95	6.34	3.21	2.54
		0.5	11.99	15.88	17.16	17.55	18.82	11.15	8.41	4.25	3.21
		0.8	15.10	18.45	17.39	18.34	18.82	13.77	9.26	4.44	3.21
		0.9	15.04	17.84	17.99	18.29	18.82	12.67	9.06	4.35	3.23

析效果低于JRipples，说明实验对JRipples产生的初始影响集产生负面影响。

为了进一步分析耦合关系相似度如何影响实验结果，我们随机从实验数据中提取一个提交的影响分析结果进行分析（如图2-5所示），我们可以发现阈值 η 取值越低，影响集中每个类的最终得分越高，说明类被调整的次数也越多， η 取值越高，则相反。这是由于阈值低的时候，能找到更多相似耦合关系对（关键类-非关键类），这同时也造成了更多的负面影响。结合图中实际影响类在不同影响集中的位置可以发现，阈值越高，实际影响类被有效调整几率越高，排名越有可能提高。但是，阈值太高时（例如 η 取0.9），由于符合相似性的耦合关系对太少，导致影响集得到的调整也变少。

综上所述，结构耦合关系的相似性可以用于映射修改模式，当相似度较低时，会对影响集产生较大调整，包括大量负面影响，当相似度过高时，则对影响集产生的调整过小。实验证，当相似度阈值 η 取值为0.8时，基于机构耦合关系相似性的历史修改模式映射取得最好效果。

2.7 本章小结

在本章中，我们提出了一种基于历史修改模式的影响分析辅助方法，该方

<i>JRipples</i>			$\eta = 0.3$			$\eta = 0.8$			$\eta = 0.9$		
排名	类名	分数	排名	类名	分数	排名	类名	分数	排名	类名	分数
1	<i>SettingsReloaded.java</i>	240	1	<i>MiscUtilities.java</i>	580	1	<i>jEdit.java</i> ↑	380	1	<i>jEdit.java</i> ↑	300
2	<i>jEdit.java</i>	260	2	<i>Log.java</i>	520	2	<i>Log.java</i>	340	2	<i>Log.java</i>	260
3	<i>MiscUtilities.java</i>	220	3	<i>EditServer.java</i>	480	3	<i>FileVFS.java</i> ↑	320	3	<i>SettingsReloaded.java</i>	240
4	<i>Log.java</i>	200	4	<i>SettingsReloaded.java</i>	460	4	<i>SettingsReloaded.java</i>	280	4	<i>MiscUtilities.java</i>	220
5	<i>FileVFS.java</i>	180	5	<i>jEdit.java</i> ↓	400	5	<i>JEditTextArea.java</i> ↑	240	5	<i>FileVFS.java</i>	180
6	<i>GUIUtilities.java</i>	160	6	<i>GUIUtilities.java</i>	340	6	<i>SearchDialog.java</i> ↑	220	6	<i>EditServer.java</i>	160
7	<i>EditServer.java</i>	140	7	<i>FileVFS.java</i> ↓	320	7	<i>MiscUtilities.java</i>	160	7	<i>GUIUtilities.java</i>	160
8	<i>VFSManager.java</i>	120	8	<i>VFSManager.java</i>	280	8	<i>GUIUtilities.java</i>	160	8	<i>VFSManager.java</i>	120
9	<i>GrabKeyDialog.java</i>	100	9	<i>GrabKeyDialog.java</i>	260	9	<i>JARClassLoader.java</i> ↑	140	9	<i>GrabKeyDialog.java</i>	100
10	<i>JARClassLoader.java</i>	80	10	<i>ToolBarManager.java</i>	260	10	<i>EditServer.java</i>	140	10	<i>JARClassLoader.java</i>	80
11	<i>VFSBrowser.java</i>	60	11	<i>JEditTextArea.java</i> ↑	240	11	<i>VFSManager.java</i>	120	11	<i>SearchDialog.java</i> ↑	80
12	<i>ToolBarManager.java</i>	40	12	<i>SearchDialog.java</i> ↓	180	12	<i>GrabKeyDialog.java</i>	100	12	<i>VFSBrowser.java</i>	60
13	<i>SearchDialog.java</i>	20	13	<i>VFSBrowser.java</i>	140	13	<i>VFSBrowser.java</i>	60	13	<i>ToolBarManager.java</i>	40
14	<i>JEditTextArea.java</i>	0	14	<i>JARClassLoader.java</i>	140	14	<i>ToolBarManager.java</i>	60	14	<i>JEditTextArea.java</i>	40

图 2-5 不同 η 下影响集优化示例

法利用开源项目的历史提交数据构建提交语料库；再借助词向量模型检索与当前修改任务的相似提交，并通过关键类判别技术识别提交中的关键类；最后，基于相似提交中关键类和非关键类之间与当前修改类与其他类之间的结构耦合关系相似性，将提交中关键类的修改模式映射回修改类，完成对初始影响集的优化。在实验中，我们在8个开源项目上进行了验证。实验结果表明我们的方法能有效提高传统影响分析的影响集的召回率和精确率。同时，我们也验证了结构耦合关系相似性对于历史修改模式映射的影响。

第3章 基于可判别特征的代码修改周期预测

软件代码修改工作通常包含多次“修改-审核-再修改”的过程，开发人员在修改完代码后需提交给审核人员，代码审核不通过，则开发人员需对代码进行再次改进。通常认为代码修改工作完成的标志是通过代码审核人员的审查。为了提高代码质量，一项代码修改任务往往经过了多次的代码审核环节，代码审核的次数直接体现该任务周期的长短。因此，我们定义代码修改的时间周期是从修改代码开始至代码通过审核并提交代码库。本文提出一种通过预测代码修改需要经过的审核次数来评估该修改时间周期的方法。代码修改周期的预测有助于开发人员及时发现代码修改中存在的问题，以及有助于项目管理人员重新评估该代码修改任务的工作量和难度，并及时做出调整，如增加开发人员等，进而缩短代码修改任务的完成周期。在本文中，我们从代码审核工具中收集开源项目维护过程中的代码审核信息，并从中提取可判别特征用于训练机器学习分类模型。我们从审核原特征（meta-feature）、代码耦合特征以及代码修改特征三个维度衡量代码修改及审核中各因素对修改周期的影响。实验中，我们在两个开源项目上验证了文中提出的方法。实验结果显示我们的预测方法具有较高的准确性。

3.1 问题描述及方法总览

代码修改周期指的是开发人员完成一项代码修改工作的时间周期，包括代码修改阶段及代码审核阶段。其中代码修改阶段所需时间主要由修改难度，修改工作量等因素的影响，代码审核阶段所需时间还受到审核过程中一些非技术因素的影响，例如审核人员、项目信息等。代码修改工作完成的标志是修改的代码经过审查人员的审核，若代码审核不通过，则将直接影响代码修改周期。在代码修改工作中提前预估修改完成周期，有助于开发人员及项目管理人员更加高效的完成代码修改工作。在本文中，我们通过代码审核次数来评估代码修改周期。我们从3000条代码审核数据中调研了代码修改工作中代码审核次数与其对应的平均修改周期的关系，如图3-1所示。从中可以发现，代码审核次数与

代码修改完成时间之间存在强相关性，代码审核次数增加意味着代码修改完成时间增加，代码修改周期的长短可以由代码审核次数反应。

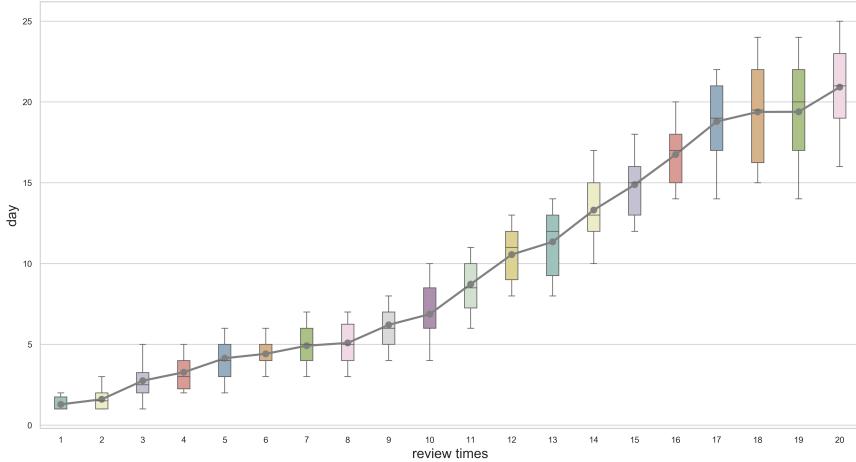


图 3-1 代码审核次数与平均修改周期关系图

图3-2展示了本文方法的总览。该方法主要分为三个阶段：代码审核数据收集阶段、可判别特征提取阶段和机器学习模型构建阶段。代码审核软件的广泛使用，使得审核过程中的历史信息更加容易收集。本文从Gerrit代码审核软件中获取开源项目维护过程中代码审核数据，主要包括代码审核信息以及代码修改信息。在可判别特征提取阶段，我们从审核原特征、代码耦合特征以及代码修改特征等三个维度提取了40维特征。审核元特征用于衡量代码审核过程中的一些非修改因素，包括审核人员信息、审核提交时间、项目信息等。代码耦合特征用于衡量修改代码实体之间的结构耦合关系，如类之间的继承关系，类与方法的调用关系等，这些结构耦合特征体现了修改工作复杂程度。代码修改特征用于衡量修改内容以及修改的工作量，如代码修改行数，所涉及方法的数量等。在模型训练阶段，本文选用在当前机器学习领域效果突出的XGBoost模型作为预测模型，并对比了多个机器学习模型的预测效果。

3.2 可判别特征的提取

本节详细介绍了修改完成周期预测模型的特征提取方法，我们将从三个维度提取特征来描述代码修改工作，这些维度特征为：审核元特征、代码耦合特

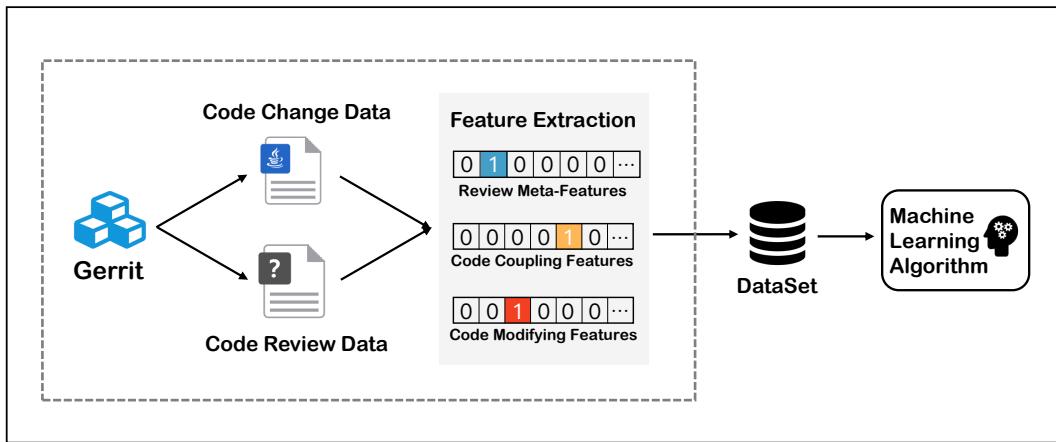


图 3-2 基于可判别特征的代码修改完成周期预测方法总览

征以及代码修改特征。审核元特征描述代码审核过程中的因素对整个时间周期的影响；代码耦合特征从修改代码的结构分析该修改工作的复杂度；代码修改特征用于衡量代码修改工作的工作量以及修改内容。

3.2.1 代码审核元特征提取

代码修改过程中的非修改因素指的是除修改代码以外的一些因素，包括负责修改代码的开发人员、代码审核人员、代码修改任务所归属的项目等，本文中，将这些非修改因素归属为审核元特征。代码修改及审核过程中存在诸多非修改因素，这些非修改因素对代码审核的通过率产生了直接的影响。审核人员的工作量和经验、开发人员的经验和参与度会直接关系到代码审核流程的质量[41]。原则上，代码审核是一个透明的过程，代码审核应该对提交代码存在的缺陷具有预防性，审核人员的目标是及时评估提交代码的质量，然而，在实践中代码审核的执行过程会受到各种因素的影响[42]。本文从代码审核数据中提取了10种特征，用于描述代码修改及审核过程中非修改因素对代码修改完成时间的影响。

本文提取的代码审核元特征如表3-1所示。 MF_0 表示该修改提交的管理人员， MF_1 表示该修改提交的作者， MF_2 表示修改提交的审核人员，我们用这三个特征来描述人力因素对完成周期的影响。通常同个软件项目下的开发人员和审核人员数量有限，不同的审核人员和开发人员对于软件项目熟悉程度的差异会导致软件修改工作所需要时间产生很大差距。同时，我们调研发现，审核人员对缺陷审查具有倾向性，有些审核人员容易忽略一些特定的代码缺陷；不同

表 3-1 代码审核文本特征及描述

类型	特征	符号	描述
审核文本特征	Owner	MF_0	修改提交的管理人员
	Author	MF_1	修改提交的作者
	Reviewers	MF_2	修改提交的审核人员
	Project	MF_3	修改所归属的项目
	Branch	MF_4	修改所归属的项目分支
	Time	MF_5	修改提交的时间
	Reviewers_Num	MF_6	审核人员的数量
	Author_Sum	MF_7	修改提交作者在项目中总的提交次数
	Author_Sum_M	MF_8	修改提交作者在项目中近一个月的提交次数
	Reviewer_Author_Ave	MF_9	相同审核人员和作者的修改的平均审核次数

审核人员提供反馈意见的详细程度也存在差异，将直接影响开发人员的再次修改。 MF_3 表示该修改提交归属的项目， MF_4 表示该修改提交所在的项目分支，我们用这两个特征衡量不同项目以及不同分支中代码修改工作的差异。不同项目以及分支中软件实体复杂程度不同，常见修改目标也不同，这些差异将体现在开发人员完成修改工作所需要的时间。另外，我们统计还发现，通常一项修改的审核工作存在多位审核人员，因此，我们收集了所有的审核人员信息，并增加了特征 MF_6 ，用于标记参与该修改提交的审核人员数量。

Eyolfson等人[65]研究了修改的提交时间与提交代码的正确性之间的关系，他们发现在深夜到凌晨4点之间提交的代码更容易出错，同时，在早上7点到中午之间提交的代码错误更少。另外，该研究还发现开发人员提交代码的频率也影响着代码的正确率。参考该研究的结论，我们从代码审核数据中提取了代码提交时间 MF_5 以及开发人员在项目中的活跃度特征，活跃度特征通过开发人员在项目中总的提交次数 MF_7 和近一个月的提交次数 MF_8 来衡量。

我们直接从审核数据中提取的特征为文本特征，如人名，项目名等。在输入预测模型前，需要将文本特征转换成数值特征，常用的编码方法有直接编码和独热编码（one-hot）。以特征 MF_2 为例，存在600多个不同的审核人员，使用直接编码的方式，将转换成一维特征和600多种取值；使用独热编码的方式，对于 MF_2 特征下600多个可能取值，将产生600多个二元特征，这些特征互斥且每次仅有一个特征激活，即每个样本仅有少数特征取值为1（由于一个提交存在多

为审核人员), 其余特征取值都为0。两种编码方式都存在明显缺陷, 审核人员之间相互独立, 而直接编码导致取值相差小的两位审核人员比取值相差大的更加具有相似性, 并且直接编码使得不同样本之间的区分度变低; 采用独热编码的方式虽然能让特征之间的距离更加合理, 但是独热编码会导致特征向量过于稀疏, 仅 MF_2 经编码后就产生600多个特征, 不易于模型训练。针对这些问题, 我们对直接编码和独热编码进行了结合和调整。如图3-3所示, 首先, 将文本类别特征按首字母划分为26类, 再对每个类下的文本进行直接编码, 这种方式增加了样本差异性并且降低了向量维度。

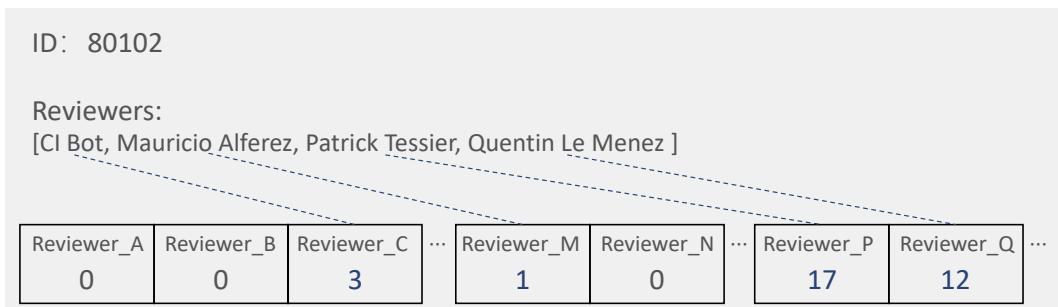


图 3-3 文本类别特征编码示例

3.2.2 代码结构耦合特征提取

软件系统经过多年的开发和维护, 软件实体间存在复杂的关联关系。由于这些关联关系, 代码修改会在软件实体间传播, 从而增加了代码修改的工作的难度。代码修改传播机制[66]表明, 当两个类之间的结构耦合关系越复杂, 则当其中一个类变更时, 另一个类存在更大的可能被影响。因此, 我们从修改代码中提取结构耦合特征, 用于描述代码修改任务的复杂度。

在代码层次, 入度和出度能直接反应一个软件实体的结构耦合特征, 入度指当前实体调用修改提交中其他实体的次数; 出度指当前实体被修改提交中其他实体调用的次数。修改所涉及的入度和出度能衡量该修改的复杂程度以及时间代价。本文从不同粒度提取软件实体的耦合特征共12维, 如表3-2所示。下面我们按不同粒度分别介绍四种耦合规则, 其中符号 C 、 M 、 A 分别表示类、方法、属性。

Class to Class: 根据面向对象编程语法, 类层次的耦合关系最常见的有继承 (Inheritance) 和实现接口 (Implementing Interface), C_i 继承 C_j 或者实现 C_j 的接

表 3-2 代码结构耦合特征及描述

类型	特征	符号	描述
代码耦合特征	Class to Class	CF_0	修改提交的类与类平均入度
		CF_1	修改提交的类与类入度总和
		CF_2	修改提交的类与类平均出度
		CF_3	修改提交的类与类出度总和
	Method to Class	CF_4	修改提交的方法与类平均入度
		CF_5	修改提交的方法与类入度总和
	Method to Attribute	CF_6	修改提交的方法与属性平均入度
		CF_7	修改提交的方法与属性入度总和
	Method to Method	CF_8	修改提交的方法与方法平均入度
		CF_9	修改提交的方法与方法入度总和
		CF_{10}	修改提交的方法与方法平均出度
		CF_{11}	修改提交的方法与方法出度总和

口, 称 C_j 为 C_i 的入度; 反之, 则称 C_j 为 C_i 的出度。本文提取四个特征用于衡量修改提交在类与类层次的耦合特征: 特征 CF_0 为当前修改提交中类的平均入度; 特征 CF_1 为当前修改提交中类的入度总和; 特征 CF_2 为当前修改提交中类的平均出度; 特征 CF_3 为当前修改提交中类的出度总和。

Method to Class: 在软件实体中, 类包含了方法集合和属性集合。类与方法的耦合关系指的是类 C_i 的某个方法 M_i 中使用了另一个类 C_j , 且不是使用 C_j 中的方法或属性, 这时类 C_j 为方法 M_i 的入度这种耦合关系常见的实例包括: 强制类型转换 (Type-Casting), 异常抛出类型 (Exception Throws), 参数类型 (Parameter Type), 返回类型 (Return Type) 和类型验证 (Instanceof) 等。本文提取两个特征衡量修改提交在方法与类层次的耦合特征: 特征 CF_4 为修改提交中所有方法 (*to Class*) 的平均入度; 特征 CF_5 为修改提交中所有方法 (*to Class*) 的入度总和。

Method to Attribute: 在面向对象编码示例中, 方法与属性层次也经常存在耦合规则, 它指的是位于类 C_i 中的某个方法 M_i 与类 C_j 中某个属性 A_j 之间的耦合关系, 其中最常见的是静态属性调用 (Static Attribute Invoking)。这种情况定义属性 A_j 为方法 M_i 的入度, 我们提取两个特征用于描述方法与属性之间的耦合关系: 特征 CF_6 表示修改提交中所有方法 (*to Attribute*) 的平均入度; 特征 CF_7 为

修改提交中所有方法（*to Attribute*）的入度总和。

Method to Method: 方法与方法层次指的是位于不同类 C_i 、 C_j 的两个方法 M_i 和 M_j 之间建立的耦合关系，满足这种耦合关系的实例有静态方法调用（Static Method Invoking），构造方法调用（Construction Method Invoking）等。这种耦合关系具有方法性，因此，我们通过四个特征来表达方法间的耦合特征：特征 CF_8 为当前修改提交中方法（*to Method*）的平均入度；特征 CF_9 为当前修改提交中方法（*to Method*）的入度总和；特征 CF_{10} 为当前修改提交中方法（*to Method*）的平均出度；特征 CF_{11} 为当前修改提交中方法（*to Method*）的出度总和。

3.2.3 代码修改特征

代码修改特征对于代码修改工作的时间代价具有很指向作用，我们从两个维度来提取修改提交中的代码修改特征，分别是代码修改量和代码修改内容。代码修改量能直接体现一项修改的工作量，而代码修改内容则是描述修改需要解决的问题和目标。代码修改量与修改内容都直接影响着代码修改阶段以及代码审核阶段所需的时间。

具体地，在代码修改量维度，可以细粒度的划分为涉及修改类的数量、修改方法的数量、修改语句的数量；而从代码变更的类型，还可以划分为增加（*add*）、修改（*change*）和删除（*delete*）。如表3-3所示，本文提取了18维特征用于描述代码修改任务中修改量。

特征 $MF_0 \sim MF_2$ 表示涉及变更的代码语句数量，包括新增代码语句的数量、修改代码语句的数量和删除代码语句的数量。特征 $MF_3 \sim MF_6$ 描述涉及修改的类的数量，分别代表新增类的数量、修改类的数量、删除类的数量以及类名修改的数量。其中类名的修改会引入一系列类引用上的修改问题。在方法层次，特征 $MF_7 \sim MF_{10}$ 分别代表方法新增的数量，方法删除的数量，方法名变更数量以及方法返回类型修改的数量。特征 $MF_{11} \sim MF_{14}$ 代表涉及变更的参数数量，包括参数类型修改，参数名修改以及参数增加和删除。特征 $MF_{15} \sim MF_{16}$ 分别表示属性类型与属性名的修改。另外，我们还发现提交表达式的修改是代码修改工作中频率较高的修改类型，并且条件语句的修改会对代码流程产生较大影响，因此我们用特征 MF_{17} 表示涉及修改的条件表达式的数

量。

表 3-3 代码修改量特征及描述

类型	符号	描述
代码修改量特征	MF_0	新增代码行数
	MF_1	修改代码行数
	MF_2	删除代码行数
	MF_3	新增类的数量
	MF_4	修改类的数量
	MF_5	删除类的数量
	MF_6	重命名类的数量
	MF_7	新增方法的数量
	MF_8	删除方法的数量
	MF_9	重命名方法数量
	MF_{10}	变更返回类型的方法数量
	MF_{11}	参数类型变更的数量
	MF_{12}	参数名变更数量
	MF_{13}	新增参数数量
	MF_{14}	删除参数数量
	MF_{15}	属性名变更数量
	MF_{16}	属性类型变更数量
	MF_{17}	条件表达式变更数量

在修改内容维度，我们采用与本文2.2节中相同的方法。我们对所有修改提交中涉及修改的代码片段进行文本预处理，再以预处理后的文本构建语料库。最后，通过Word2vec方法训练词向量模型，以50维的特征向量表示修改提交中的修改内容。

3.3 机器学习算法选择及模型评估方法

本文提出的代码修改周期预测方法基于有监督机器学习算法，具体地，我们通过代码修改的审核次数评估代码修改的周期长短。我们将代码修改周期预测视为分类问题，在实验预测中，审核次数被划分为三个区间，分别对应代码修改周期的较短、一般以及较长三个层次。我们从开源项目的历史修改提交数据中提取审核元特征，代码耦合特征以及代码修改特征，再基于这些特征训练

机器学习模型。最后使用训练好的模型对新的修改任务预测修改周期。

3.3.1 机器学习算法选择

在本文中，采用有监督机器学习算法，其中，常见的分类算法有逻辑回归（*LR*, Logistic Regression），支持向量机（*SVM*, Support Vector Machines），随机森林（*RF*, Random Forest），人工神经网络（*ANN*, Artificial Neural Networks）以及*XGBoost*（Extreme Gradient Boosting）。在实验中，我们通过对比各种分类算法在两个开源项目中的分类效果，选择表现最优的*XGBoost* 作为我们的分类模型。

XGBoost 是梯度提升树模型（Gradient Boosting Trees）的一种高效实现方式。*XGBoost* 集成了K颗*CART*树（Classification and Regression Trees） $\{Tr_1(x_1, y_1), Tr_2(x_2, y_2) \dots, Tr_k(x_k, y_k)\}$ 。其中， x_i 表示第 i^{th} 个样本的特征， y_i 则表示该样本对应的标签。*CART*树将分配给每个叶子节点一个分数，而最终的预测结果将由所有*CART*树上对应叶子节点的分数通过加法模型得到，如公式所示3.1。

$$\hat{y} = \sum_{k=1}^K f_k(x_i), \quad f_k \in F \quad (3.1)$$

在上述公式中， $f_k(x_i)$ 为第 k^{th} 颗树的预测分数， F 则代表函数空间中所包含的所有树模型。相比于传统的梯度提升树模型，*XGBoost*一个重要的改进是在目标函数中加入了正则项。公式3.2为*XGBoost*的目标函数。

$$Obj(\Theta) = \sum_{i=1}^n l(y_i, \hat{y}_i) + \sum_{k=1}^K \Omega(f_k) \quad (3.2)$$

其中， l 为损失函数，描述的是预测标签 \hat{y}_i 与真实标签 y_i 之间的误差。而在表达式 $\sum_k \Omega(f_k)$ 中， $\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^T \omega_j^2$ 用于描述模型的复杂度。 T 和 ω 分别表示叶子数量及对应叶子节点上的取值。 γ 和 λ 则用于控制正则化的惩罚程度。

3.3.2 模型评估方法

为了评估修改周期预测模型的有效性，我们采用了分类模型评价中常用的四个指标：召回率、精确率、准确率以及 F_1 值。召回率用于度量模型对特定类

别样本的预测安全性，如对一些修改周期较长的样本，是否能准确预测；精确率用于度量分类模型的预测结果的精确性；而准确率及 F_1 则考察模型对多个类别预测的综合效果。四个指标的定义如下：

$$precision = \frac{TP}{TP + FP} \quad (3.3)$$

$$recall = \frac{TP}{TP + FN} \quad (3.4)$$

$$F_1 = 2 * \frac{precision * recall}{precision + recall} \quad (3.5)$$

$$accuracy = \frac{TP + TN}{TP + FN + TN + FP} \quad (3.6)$$

以二分类为例（多分类视为多个二分类问题）， TP 和 TN 分别表示正确预测的正样本数量和负样本数量， FP 和 FN 分别表示错误预测的正样本数量和负样本数量。

3.4 代码修改周期预测实验设置与评估

本节首先描述了实验数据的收集及去噪处理方法，然后介绍了在修改周期预测实验中关注的三个研究问题，最后分析了模型预测的实验结果。

3.4.1 数据收集及去噪处理

我们从Gerrit代码审核工具中收集开源项目的历史提交信息以及代码审核信息。我们选择了Eclipse^①和OpenDaylight^②两个项目作为实验数据集。其中，Eclipse是使用最广泛的集成开发环境之一，而OpenDaylight是一种高度可用、模块化以及可扩展的多协议控制器基础架构，用于供应商网络上的SDN部署。Eclipse和OpenDaylight都拥有丰富的代码修改历史数据，可用于代码修改周期的分析。表格展示了实验数据集的详细信息。

另外，我们对大量代码修改数据进行观察，发现其中存在一些噪声数据，这些数据对模型的训练产生负面影响。我们设置了一下几条规则对噪声数据进行过滤：

^① <https://www.eclipse.org/>

^② <https://www.opendaylight.org/>

表3-4 代码修改周期预测实验数据集

Projects	Time span	No. of reviews
Eclipse	2016.1-2018.5	9455
OpenDaylight	2017.1-2018.8	6403

- 1) 在代码修改周期的预测中，我们需要从被修改的代码中提取代码修改特征及代码耦合特征。许多提交中不涉及代码修改，我们将过滤这部分的提交数据。
- 2) 我们同时去除数据集中，明显存在异常的修改提交。例如，有一个修改提交需要经过150次的反复修改和审核，而所有修改数据的平均修改和审核次数为5次。
- 3) 另外，我们还去除了项目中不活跃的开发人员的修改提交。我们认为修改提交次数低于两次的开发人员为项目中的非核心维护人员，这些开发人员只在项目中完成极少数的代码修改工作。

3.4.2 代码修改周期预测的问题研究

文章的主要目的是从代码修改及代码审核两个阶段分析代码修改周期，试着通过代码修改及审核次数预测代码修改周期的长短。本节从多个角度研究，评估本文提出的代码修改周期预测方法的有效性。我们关注的研究问题有以下三个：

研究问题1：本文方法对代码修改任务能否在较短周期内完成的预测效果如何？

开发人员都希望代码修改任务能在较短的时间周期内完成，即一次修改和审核周内，其中关键的是代码修改不存在问题并且在代码审核过程被接受。值得注意的是，在这个研究问题中，我们将审核次数为1的提交数据视为正样本，其余提交数据视为负样本。因此，我们使用本文提出的方法，训练预测模型，预测代码修改任务的审核次数是否为1。数据集中正负样本分布如表3-5所示。

研究问题2：本文方法对代码修改周期的长短的预测效果如何？

表 3-5 研究问题1的样本分布

Projects	Positive	Negative
Eclipse	2905	6551
OpenDaylight	1598	4807

观察图***可以发现，一项代码修改任务所花费时间周期的长短可以根据代码修改和审核次数来划分。在这个研究问题中，我们通过划分审核次数区间，来对应代码修改周期长短的层次。审核次数为1的提交的平均修改周期为1天，我们将这类修改对应为“较短”的代码修改周期；审核次数在2-6区间的提交的平均修改周期为1周内，我们认为这类修改任务对应代码修改周期长短为“一般”；而审核次数在7-20区间的提交的平均修改周期大于一周，我们认为这类修改任务需要“较长”的修改周期。经过划分后的数据分布如表所示

表 3-6 研究问题2的样本分布

Projects	较短	一般	较长
Eclipse	2905	5203	1287
OpenDaylight	1598	3327	1338

研究问题3：三种类型的特征对模型预测效果的影响如何？

本文方法的核心是从代码修改的历史信息中提取审核元特征、代码耦合特征及代码修改特征，用于度量代码修改的时间周期长短。三种特征代表了代码修改阶段及代码审核阶段对代码修改周期的影响。在这个研究问题中，我们关注于三种类型各自的特征对模型预测效果的影响。

3.4.3 实验结果与分析

研究问题1：本文方法对代码修改任务能否在较短周期内完成的预测效果如何？

表格3-7展示了本文方法对代码修改任务能否在较短周期内完成的预测效果，包括正负样本的召回率、精确率和 F_1 值，以及综合预测效果的准确率。在实验

表 3-7 研究问题1实验结果

Projects	Method	Positive Instances			Negative Instances			<i>Accuracy</i>
		<i>precision</i>	<i>recall</i>	<i>F₁</i>	<i>precision</i>	<i>recall</i>	<i>F₁</i>	
Eclipse	LR	69.39	33.72	45.38	74.84	92.98	82.93	73.99
	SVM	69.89	42.98	53.22	77.24	91.27	83.67	75.79
	RF	76.85	40.00	52.55	76.85	94.23	84.69	76.85
	ANN	63.21	49.42	55.47	78.37	86.44	82.21	74.58
	XGBoost	71.63	54.80	62.10	82.16	90.56	86.16	79.72
OpenDaylight	LR	60.47	33.66	43.24	78.36	78.36	84.66	74.86
	SVM	41.38	19.41	26.43	77.52	90.10	83.72	73.34
	RF	51.75	43.04	46.99	82.32	86.86	84.53	76.06
	ANN	49.18	9.71	16.22	76.59	96.71	85.49	75.26
	XGBoost	50.39	50.78	50.58	87.57	87.40	87.49	80.03

中，我们测试对比了本文提出方法在多个机器学习模型上的效果，并采用10折交叉验证方法。针对数据集样本分布不平衡问题，使用SMOTE算法生成训练集中占比较低的样本，测试集中不做改变。

观察表3-7可以发现，在两个数据集上不同机器学习模型的预测准确率分布为73.34%至80.03%，这体现了本文提出方法对代码修改任务能否在较短周期内完成的预测的准确性。对比不同机器学习模型的效果，可以发现XGBoost取得了最优的表现。在Eclipse数据集上，XGBoost对负样本预测的召回率、精确率和*F₁*值分别是82.16%，90.56%和86.16%，意味着大部分负样本都能被准确预测。负样本代表可能需要重复多次修改的提交，对该类样本的预测有助于开发人员和项目管理提前做出调整和安排。另一方面，在正样本的预测效果上，多种机器学习模型的效果都要低于在负样本上表现。我们认为原因是SMOTE算法并不能完全解决样本不平衡的问题。

另外，我们可以发现XGBoost 的负样本召回率要稍微低于其他机器学习模型，但是，从正负样本的综合预测效果来看，XGBoost 的表现还是明显优于其他机器学习算法。因此，我们选择XGBoost 作为我们的预测模型。

研究问题2：本文方法对代码修改周期的长短的预测效果如何？

表格3-8是本文方法对代码修改周期的长短的预测结果。在研究问题2中，我们同样对比了多种机器学习模型，以及通过SMOTE算法来一定程度样本不平衡造成的影响。观察实验结果可以发现，XGBoost 的综合准确在两个数据集上

表 3-8 研究问题2实验结果

Projects	Method	较短			一般			较长			Accuracy
		precision	recall	F ₁	precision	recall	F ₁	precision	recall	F ₁	
Eclipse	LR	66.67	22.64	33.79	61.20	60.90	61.05	29.70	74.24	42.42	50.72
	SVM	64.04	43.92	52.10	64.07	57.18	60.04	33.57	71.21	45.63	54.98
	RF	60.00	54.73	57.24	64.96	72.14	68.36	46.31	35.61	40.26	61.52
	ANN	63.81	40.20	49.33	62.57	72.73	67.27	41.32	49.62	45.09	59.23
	XGBoost	69.59	54.90	61.38	66.45	80.94	72.98	57.23	35.99	44.19	66.42
OpenDaylight	LR	52.14	23.62	32.52	59.73	61.68	60.69	44.22	65.89	52.93	53.31
	SVM	44.48	23.95	31.03	57.58	65.11	61.11	47.08	55.96	51.13	52.75
	RF	48.57	44.01	46.18	59.10	69.31	63.80	54.09	39.40	45.59	55.87
	ANN	55.10	17.47	26.54	57.88	62.31	60.01	43.10	66.23	52.22	52.19
	XGBoost	60.10	37.54	46.22	61.03	77.57	68.31	58.61	47.35	52.38	60.42

分别达到66.41% 和60.42%， 高于其他机器学习模型的预测效果。同时，我们发现，所有机器学习模型的综合准确率分布为50.72%至66.42%， 表明了我们方法在代码修改周期预测上的有效性。在Eclipse数据集上，XGBoost对于不同时间周期的F₁值分别达到61.38%， 72.68%， 44.19%。而在OpenDaylight上的效果则有一些下降，我们认为原因是OpenDaylight数据集中样本低于Eclipse。

另外，我们发现XGBoost在修改周期“较长”的样本上，召回率要略低于一些机器学习算法，但从综合评价指标F₁值可以发现，XGBoost 在该研究问题中是效果最好且稳定的模型。

研究问题3：三种类型的特征对模型预测效果的影响如何？

为了研究三种特征对模型预测效果的影响，我们在研究问题1上进行对比实验。具体地，我们对三种特征进行两两组合，分别测试每对特征训练出来的模型的预测效果。例如，若审核元特征与代码耦合特征的组合训练出来的模型预测效果最优，则说明审核元特征与代码耦合特征对预测结果的重要性要高于代码修改特征。在实验中，我们对比了不同特征组合下模型预测效果的F₁值和准确率，实验结果如表3-9所示。

观察表3-9中数据可以发现，任意两种特征组合的预测效果都低于使用三种特征时的预测效果，这说明三种对模型预测都起着重要作用。同时也证明了本文提取的三种可判别特征类型可用于代码修改周期的预测。使用审核元特征与代码耦合特征的组合训练得到的模型在F₁值和准确率上都低于其他两组特征组

表 3-9 Results of RQ3

Projects	Features	F_1		<i>Accuracy</i>
		Positive Instances	Negative Instances	
Eclipse	审核元特征&代码耦合特征	48.31	83.77	75.30
	审核元特征&代码修改特征	51.04	85.17	77.24
	代码耦合特征&代码修改特征	54.78	85.08	77.56
	所有特征	62.10	86.16	79.72
OpenDaylight	审核元特征&代码耦合特征	42.86	83.67	74.60
	审核元特征&代码修改特征	35.90	85.38	76.19
	代码耦合特征&代码修改特征	44.09	85.88	77.46
	所有特征	50.58	87.49	80.03

合，意味着代码修改特征在修改周期预测中的重要性高于另外两种特征。另一方面，代码耦合特征与代码修改特征训练的模型取得了最高的 F_1 值和准确率，这表明审核元特征在周期预测中的作用低于另外两种特征。

3.5 本章小结

软件维护中的代码修改工作通常包括了代码修改与代码审核，且往往需要“修改-审核”多次才能最终完成。本文以“代码修改-代码审核”的次数衡量一项代码修改工作的时间周期长短，从软件项目的历史代码修改信息中，提取可判别特征，再基于机器学习算法训练预测模型。该模型对一项代码修改工作预测其可能需要经过“修改-审核”的次数，再以次数对应其修改周期的长短。实验中，我们在开源项目Eclipse和OpenDaylight上验证了我们方法的有效性。实验结果表明，本文方法提取的可判别性特征可以用于预测代码修改工作的时间周期。

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。

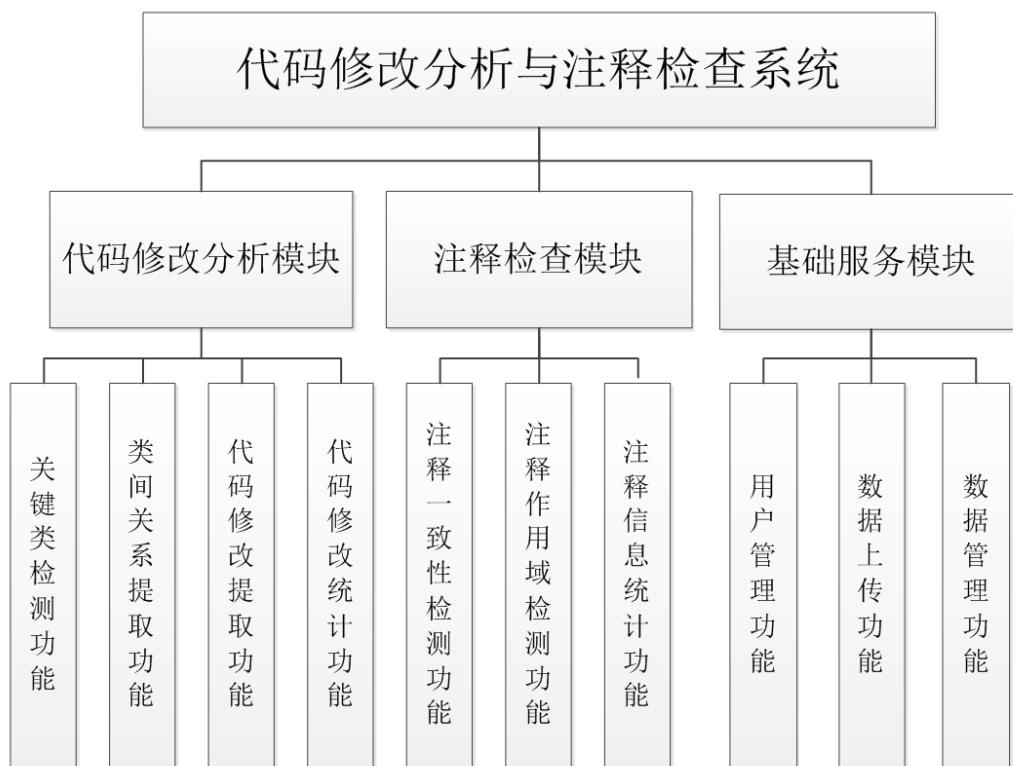


图 4-1 系统功能结构图

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)。

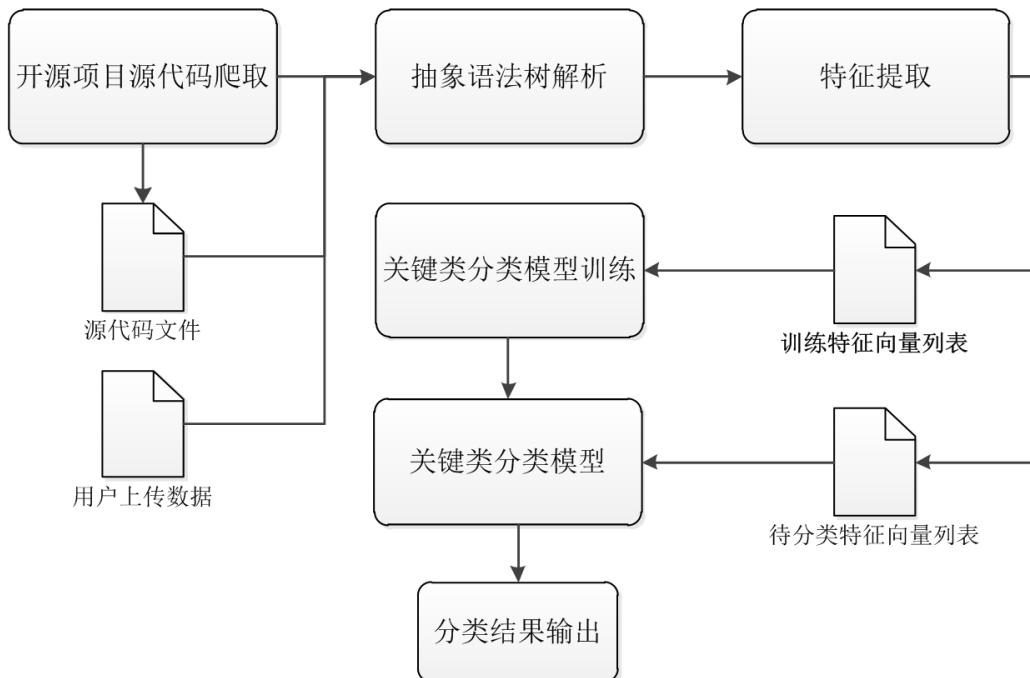


图 4-2 关键类分类模型构建过程

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，

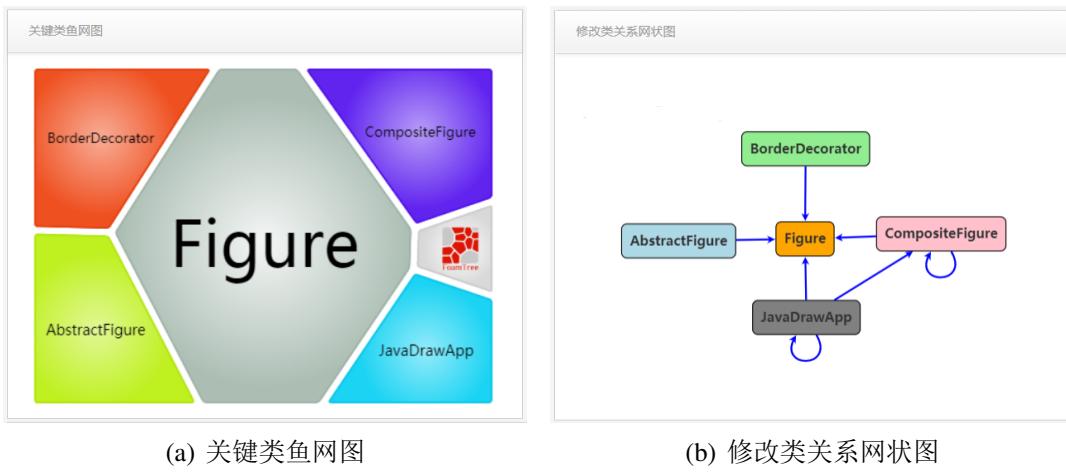


图 4-3 关键类鱼网图和修改类关系网状图

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

代码修改统计信息

核心类分析

注释统计信息

注释一致性检测

用户上传列表

关键类鱼网图

修改类关系网状图

```

graph TD
    BD[BorderDecorator] --> F[Figure]
    AF[AbstractFigure] --> F
    CF[CompositeFigure] --> F
    JD[JavaDrawApp] --> F
    JD --> AF
    JD --> CF
    JD -.-> BD
  
```

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      * @see FigureDecorator#draw
93     */
94     /**
95      * This method draws the FigureDecorators
96      *
97      * @see FigureDecorator#draw
98     */
99     public void drawAll(Graphics g) {
100
101    /**
102     * Returns the handles used to manipulate
103     * the figure. Handles is a Factory Method for
104     * creating handle objects.
105     *
106     * @return an type-safe iterator of handles
107     * @see Handle
108    */
109    public HandleEnumeration handles();
110
111    /**
112     * Gets the size of the figure
113    */
114    public Dimension size();
115
116    /**
117     * Gets the figure's center
118    */
119    public Point center();
120
121    /**
122     * Checks if the Figure should be considered as empty.
123     * For instance, when the figure is being added to the drawing, the <code>
124     * CreationTool</code> will check this to see whether to add the figure or
125     * not. If the figure is too small to be seen, it will return true for isEmpty.
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
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
  
```

图 4-4 修改类信息展示页面

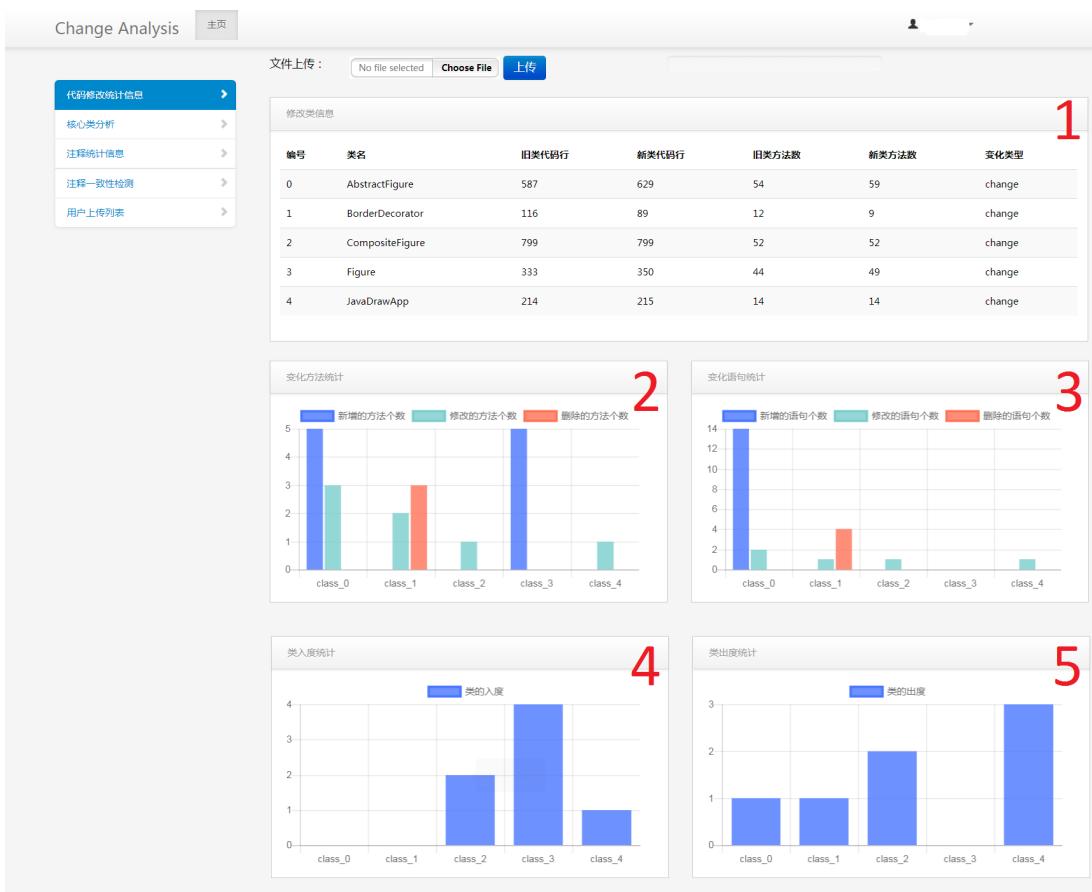


图 4-5 代码修改统计页面

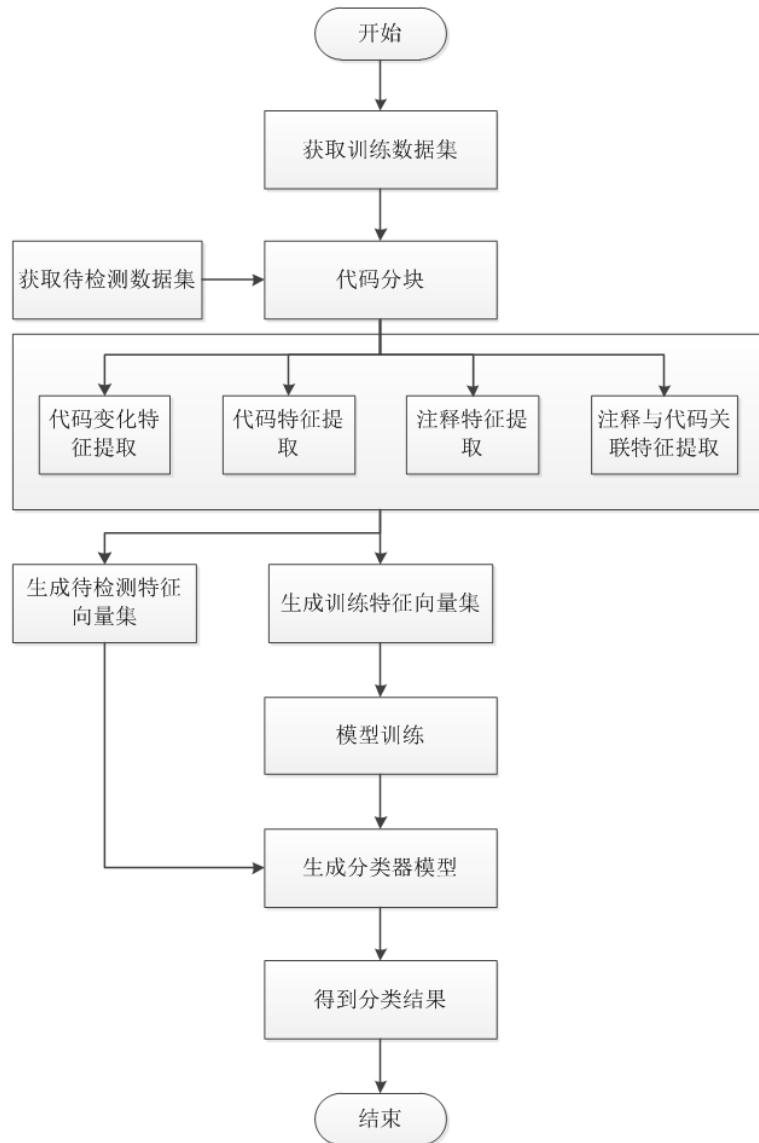


图 4-6 注释一致性检测流程图

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

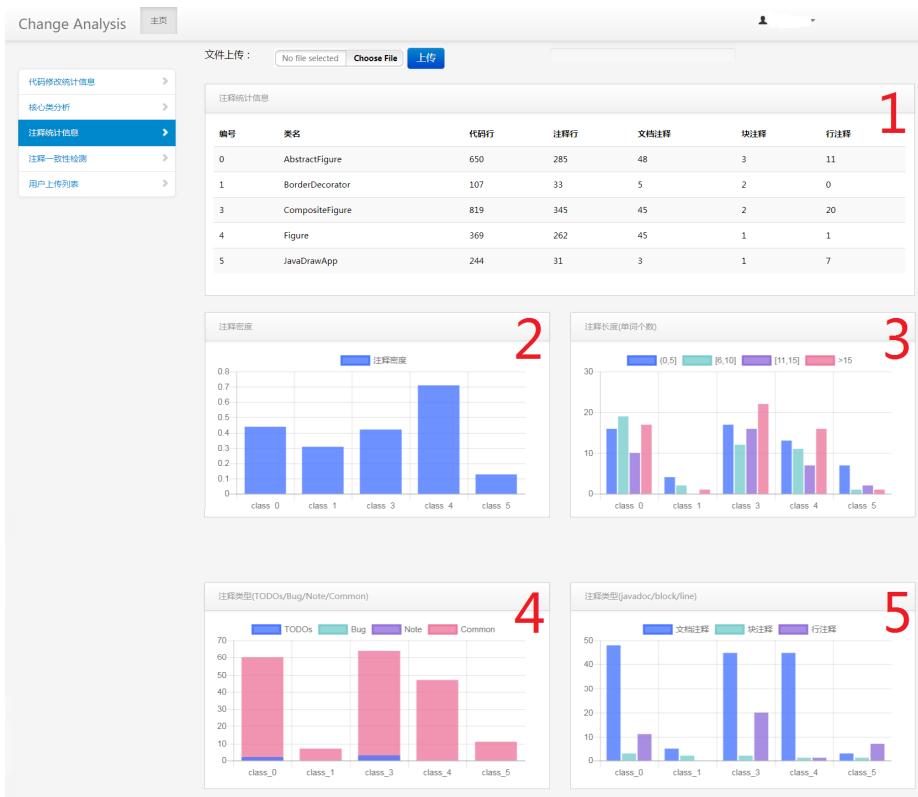


图 4-8 注释信息统计页面

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] Claudia Szabo. Novice code understanding strategies during a software maintenance assignment. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 276–284. IEEE Press, 2015.
- [2] Per Rovegård, Lefteris Angelis, and Claes Wohlin. An empirical study on views of importance of change impact analysis issues. *IEEE Transactions on Software Engineering*, (4):516–530, 2008.
- [3] Sunghun Kim and Michael D Ernst. Prioritizing warning categories by analyzing software history. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 27. IEEE Computer Society, 2007.
- [4] Alexander WJ Bradley and Gail C Murphy. Supporting software history exploration. In *Proceedings of the 8th working conference on mining software repositories*, pages 193–202. ACM, 2011.
- [5] Moritz Wittenhagen, Christian Cherek, and Jan Borchers. Chronicler: Interactive exploration of source code history. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 3522–3532. ACM, 2016.
- [6] Marco D’ Ambros, Harald Gall, Michele Lanza, and Martin Pinzger. Analysing software repositories to understand software evolution. In *Software evolution*, pages 37–67. Springer, 2008.
- [7] Stephen W Thomas, Bram Adams, Ahmed E Hassan, and Dorothea Blostein. Studying software evolution using topic models. *Science of Computer Programming*, 80:457–479, 2014.
- [8] Georgios Gousios, Eirini Kalliamvakou, and Diomidis Spinellis. Measuring developer contribution from software repository data. In *Proceedings of the 2008 in-*

- ternational working conference on Mining software repositories*, pages 129–132. ACM, 2008.
- [9] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim 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*, pages 1277–1286. ACM, 2012.
- [10] Gerardo Canfora and Luigi Cerulo. Impact analysis by mining software and change request repositories. In *Software Metrics, 2005. 11th IEEE International Symposium*, pages 9–pp. IEEE, 2005.
- [11] Huzefa Kagdi, Malcom Gethers, Denys Poshyvanyk, and Michael L Collard. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 119–128. IEEE, 2010.
- [12] Xiaobing Sun, Bixin Li, Hareton Leung, Bin Li, and Yun Li. Msr4sm: Using topic models to effectively mining software repositories for software maintenance tasks. *Information and Software Technology*, 66:1–12, 2015.
- [13] Edmund Wong, Taiyue Liu, and Lin Tan. Clocom: Mining existing source code for automatic comment generation. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 380–389. IEEE, 2015.
- [14] Yuan Huang, Qiaoyang Zheng, Xiangping Chen, Yingfei Xiong, Zhiyong Liu, and Xiaonan 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*, pages 414–423. IEEE Press, 2017.
- [15] Ahmet Okutan and Olcay Taner Yıldız. Software defect prediction using bayesian networks. *Empirical Software Engineering*, 19(1):154–181, 2014.

- [16] Julie Moeyersoms, Enric Junqué de Fortuny, Karel Dejaeger, Bart Baesens, and David Martens. Comprehensible software fault and effort prediction: A data mining approach. *Journal of Systems and Software*, 100:80–90, 2015.
- [17] Stacy Nelson and Johann Schumann. What makes a code review trustworthy? In *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*, pages 10–pp. IEEE, 2004.
- [18] Anne Edmundson, Brian Holtkamp, Emanuel Rivera, Matthew Finifter, Adrian Mettler, and David Wagner. An empirical study on the effectiveness of security code review. In *International Symposium on Engineering Secure Software and Systems*, pages 197–212. Springer, 2013.
- [19] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):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, 42(12):2467–2476, 2014.
- [21] Salah Bouktif, Yann-Gael Gueheneuc, and Giuliano Antoniol. Extracting change-patterns from cvs repositories. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 221–230. IEEE, 2006.
- [22] Shawn A Bohner and RS Arnold. Software change impact analysis for design evolution. In *Proceedings of 8th International Conference on Maintenance and Re-engineering*, pages 292–301. IEEE CS Press Los Alamitos, CA, 1991.
- [23] Seunghun Park and Doo-Hwan Bae. An approach to analyzing the software process change impact using process slicing and simulation. *Journal of Systems and Software*, 84(4):528–543, 2011.
- [24] Huzefa Kagdi, Malcom Gethers, and Denys Poshyvanyk. Integrating conceptual and logical couplings for change impact analysis in software. *Empirical Software Engineering*, 18(5):933–969, 2013.

- [25] Hani Abdeen, Khaled Bali, Houari Sahraoui, and Bruno Dufour. Learning dependency-based change impact predictors using independent change histories. *Information and Software Technology*, 67:220–235, 2015.
- [26] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *Proceedings of the 27th international conference on Software engineering*, pages 432–441. ACM, 2005.
- [27] Lulu Huang and Yeong-Tae 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*, pages 374–384. IEEE, 2007.
- [28] Chin-Yu Lin, Tung-Yueh Wu, and Chin-Cheng Huang. Nonlinear dynamic impact analysis for installing a dry storage canister into a vertical concrete cask. *International Journal of Pressure Vessels and Piping*, 131:22–35, 2015.
- [29] Haipeng Cai and Douglas Thain. Distia: A cost-effective dynamic impact analysis for distributed programs. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 344–355. ACM, 2016.
- [30] Ben Breech, Mike Tegtmeyer, and Lori Pollock. Integrating influence mechanisms into impact analysis for increased precision. In *Software Maintenance, 2006. IC-SM’06. 22nd IEEE International Conference on*, pages 55–65. IEEE, 2006.
- [31] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 308–318. IEEE Computer Society, 2003.
- [32] Haipeng Cai and Raul Santelices. Diver: Precise dynamic impact analysis using dependence-based trace pruning. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 343–348. ACM, 2014.

- [33] Lionel Briand, Yvan Labiche, and George Soccar. Automating impact analysis and regression test selection based on uml designs. In *icsm*, page 0252. IEEE, 2002.
- [34] Jessica Díaz, Jennifer Pérez, Juan Garbajosa, and Alexander L Wolf. Change impact analysis in product-line architectures. In *European Conference on Software Architecture*, pages 114–129. Springer, 2011.
- [35] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical software engineering*, 14(1):5–32, 2009.
- [36] Arpad Beszedes, Tamas Gergely, Szabolcs Farago, Tibor Gyimothy, and Ferenc Fischer. The dynamic function coupling metric and its use in software evolution. In *Software Maintenance and Reengineering, 2007. CSMR’07. 11th European Conference on*, pages 103–112. IEEE, 2007.
- [37] Thomas Rolfsnes, Stefano Di Alesio, Razieh Behjati, Leon Moonen, and Dave 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)*, pages 201–212. IEEE, 2016.
- [38] Lile Hattori, Gilson dos Santos Jr, Fernando Cardoso, and Marcus Sampaio. Mining software repositories for software change impact analysis: a case study. In *Proceedings of the 23rd Brazilian symposium on Databases*, pages 210–223. Sociedade Brasileira de Computação, 2008.
- [39] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 23(8):613–646, 2013.
- [40] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi 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*, pages 141–150. IEEE, 2015.
- [41] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yixin Cao, and Michael W Godfrey. Investigating code review quality: Do people and participation matter? In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 111–120. IEEE, 2015.
- [42] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W Godfrey. Investigating technical and non-technical factors influencing modern code review. *Empirical Software Engineering*, 21(3):932–959, 2016.
- [43] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Bayesian specification learning for finding api usage errors. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 151–162. ACM, 2017.
- [44] Chris Mills. Automating traceability link recovery through classification. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 1068–1070. ACM, 2017.
- [45] Michael Rath, Jacob Rendall, Jin LC Guo, Jane Cleland-Huang, and Patrick Mäder. Traceability in the wild: automatically augmenting incomplete trace links. In *Proceedings of the 40th International Conference on Software Engineering*, pages 834–845. ACM, 2018.
- [46] Syaeful Karim, Harco Leslie Hendric Spits Warnars, Ford Lumban Gaol, Edi Abdurachman, Benfano 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*, pages 19–23. IEEE, 2017.
- [47] Kento Shimonaka, Soichi Sumi, Yoshiki Higo, and Shinji Kusumoto. Identifying auto-generated code by using machine learning techniques. In *Empirical Software*

- Engineering in Practice (IWESEP), 2016 7th International Workshop on*, pages 18–23. IEEE, 2016.
- [48] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien 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*, pages 457–468. ACM, 2014.
- [49] Mario Linares-Vásquez, Collin McMillan, Denys Poshyvanyk, and Mark Grechanik. On using machine learning to automatically classify software applications into domain categories. *Empirical Software Engineering*, 19(3):582–618, 2014.
- [50] Annie TT Ying, Gail C Murphy, Raymond Ng, and Mark C Chu-Carroll. Predicting source code changes by mining change history. *IEEE transactions on Software Engineering*, 30(9):574–586, 2004.
- [51] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.
- [52] Nemitari Ajienka, Andrea Capiluppi, and Steve Counsell. An empirical study on the interplay between semantic coupling and co-change of software classes. *Empirical Software Engineering*, 23(3):1791–1825, 2018.
- [53] Fehmi Jaafar, Yann-Gaël Guéhéneuc, Sylvie Hamel, and Giuliano Antoniol. Detecting asynchrony and dephase change patterns by mining software repositories. *Journal of Software: Evolution and Process*, 26(1):77–106, 2014.
- [54] Stéphane Vaucher, Houari Sahraoui, and Jean Vaucher. Discovering new change patterns in object-oriented systems. In *Reverse Engineering, 2008. WCRE’08. 15th Working Conference on*, pages 37–41. IEEE, 2008.

- [55] Beat Fluri, Emanuel Giger, and Harald C Gall. Discovering patterns of change types. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 463–466. IEEE Computer Society, 2008.
- [56] Amir Michail. Data mining library reuse patterns in user-selected applications. In *ase*, page 24. IEEE, 1999.
- [57] Yu Lin and Danny Dig. Check-then-act misuse of java concurrent collections. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 164–173. IEEE, 2013.
- [58] Reid Holmes, Robert J Walker, and Gail C Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, (12):952–970, 2006.
- [59] Marcel Bruch, Martin Monperrus, and Mira 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*, pages 213–222. ACM, 2009.
- [60] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie Van Deursen. Mining software repositories to study co-evolution of production & test code. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 220–229. IEEE, 2008.
- [61] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering*, 33(11), 2007.
- [62] Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.’s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

- [63] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.
- [64] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.
- [65] Jon Eyolfson, Lin Tan, and Patrick Lam. Do time of day and developer experience affect commit bugginess? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 153–162. ACM, 2011.
- [66] Ahmed E Hassan and Richard C Holt. Predicting change propagation in software systems. In *Software maintenance, 2004. proceedings. 20th ieee international conference on*, pages 284–293. IEEE, 2004.

致谢