软件学报 ISSN 1000-9825, CODEN RUXUEW E-mail: jos@iscas.ac.cn
Journal of Software, [doi: 10.13328/j.cnki.jos.000000] http://www.jos.org.cn

Tel: +86-10-62562563

即时软件缺陷预测研究进展*

蔡 亮[1], 范元瑞[1], 鄢 萌[1], 夏 鑫[2]

[1](浙江大学, 计算机科学与技术学院, 浙江 杭州 310007)

[2](Monash University, Australia Melbourne VIC 3800)

通讯作者: 鄢萌, E-mail: mengy@zju.edu.cn

摘 要: 软件缺陷预测一直是软件工程研究中最活跃的领域之一，研究人员已经提出了大量的缺陷预测技术，根据预测粒度不同，主要包括模块级、文件级和变更级(Change-level)缺陷预测。其中变更级缺陷预测旨在于开发者提交代码时，对其引入的代码是否存在缺陷进行预测，因此又被称作即时(Just-in-time)缺陷预测。近年来，即时缺陷预测技术由于其即时性、细粒度等优势，成为了缺陷预测领域的研究热点，取得了一系列研究成果，同时也在数据标注、特征提取、模型评估等环节面临诸多挑战，迫切需要更先进、统一的理论指导和技术支撑。鉴于此，本文从即时缺陷预测技术的数据标注、特征提取和模型评估等方面对近年来即时缺陷预测研究进展进行梳理和总结。主要内容包括：(1)归类并梳理了即时缺陷预测模型构建中数据标注常用方法及其优缺点；(2)对即时缺陷预测的特征类型和计算方法进行了详细分类和总结；(3)总结并归类现有模型构建技术；(4)总结了模型评估中使用的实验验证方法与性能评估指标；(5)归纳出了即时缺陷预测技术的关键问题；(6)最后展望了即时缺陷预测的未来发展。

关键词: 软件缺陷预测; 即时缺陷预测; 软件维护; 软件质量; 软件工程

中图法分类号: TP311

中文引用格式: 蔡亮,范元瑞,鄢萌,夏鑫. 即时软件缺陷预测研究进展.软件学报. http://www.jos.org.cn/1000-9825/ 0000.htm

英文引用格式: Cai L, Fan YR, Yan M, Xia X. Just-in-time software defect prediction: A road map. Ruan Jian Xue Bao/Journal of Software, 2018 (in Chinese). http://www.jos.org.cn/1000-9825/0000.htm

Just-in-Time Software Defect Prediction: A Road Map

CAI Liang[1], FAN Yuan-Rui[1], YAN Meng[1], XIA Xin[2]

[1](College of Computer Science and Technology, Zhejiang University, Hangzhou 310007, China)

[2](Faculty of Information Technology, Monash University, Melbourne, VIC 3800, Australia)

Abstract: Software Defect prediction is always one of the most active research areas in software engineering. Researchers have proposed a lot of defect prediction techniques. These techniques consist of module-level, file-level and change-level defect prediction according to the granularity. Change-level defect prediction can predict the defect-proneness of changes when they are initially submitted. Hence, such a technique is referred to as Just-in-Time defect prediction. Recently, Just-in-Time defect prediction becomes the hot area in defect prediction because of its timely manner and fine-grained. There are a lot of achievements in this area and there are also many challenges in data labeling, feature extracting and model evaluation. More advanced and unified theoretic and technica1 guidelines are needed to enhance Just-in-Time defect prediction. Therefore, in this paper, we present a road map for prior Just-in-Time defect prediction studies in three folds, data labeling, feature extraction and model evaluation. In summary, the contributions of this paper are: (1) We

* 基金项目: 国家自然科学基金(00000000, 00000000);
Foundation item: National Natural Science Foundation of China (00000000, 00000000);
收稿时间: 0000-00-00; 修改时间: 0000-00-00; 采用时间: 0000-00-00; jos 在线出版时间: 0000-00-00
CNKI 在线出版时间: 0000-00-00

conclude the data labeling methods and their advantages and disadvantages. (2) We conclude and classify the feature categories and computing methods. (3) We conclude and classify the modeling techniques. (4) We conclude the model validation and performance measures in model evaluation. (5) We highlight the current problems in this area. (6) We conclude the trends of Just-in-Time defect prediction.

Key words: Software defect prediction; Just-in-Time defect prediction; Software maintenance; Software quality; Software engineering

1 引言

研究表明，由于软件系统的复杂性，软件缺陷常常不可避免。根据 IEEE 标准定义，软件缺陷是指软件产品中存在的，导致产品无法满足软件需求和其规格要求，需要进行修复的瑕疵、问题[1]。软件缺陷的存在极大制约了软件的应用与发展，带来了大量经济损失。据美国国家标准与技术研究院 (National Institute of Standards and Technology, NIST)估算，由于软件缺陷给美国带来的一年经济损失高达 600 亿美金，NIST 通过进一步研究发现，识别和修复这些软件缺陷，能帮助美国节省 220 亿美金[2]。因此，修复缺陷成为了软件维护中的关键活动，但它同时需要消耗大量的时间和资源[3]。数据表明，修复缺陷所需要的成本占软件开发总成本的 50-75%[4]。及时的发现缺陷有助于减少修复缺陷的成本，提高软件质量。

鉴于此，软件工程领域的研究者提出了软件缺陷预测技术，其目的在于提前预测可能存在缺陷的软件实体(如文件和变更)[5]。其基本思路在于从软件项目的历史数据中提取特征用来表征被预测的软件实体，然后将这些特征输入到分类(或回归)器进行训练获得预测模型，从而对新产生的软件实体预测其存在缺陷的可能性[6]。其研究意义在于：

(1) **优化资源分配，节省维护成本**。缺陷预测技术可以对软件的文件、模块等计算其存在缺陷的可能性，从而实现对有限开发资源的合理调度，例如，对更有可能出现缺陷的文件和模块进行更多审查和测试工作[7];

(2) **及时发现缺陷，提高代码质量。**使用缺陷预测技术，能帮助开发人员及时发现潜在缺陷，及时采取缺陷修复措施，有助于更好的保证软件质量[8];

软件缺陷预测研究有超过 40 年的历史[9]。传统缺陷预测技术主要针对粗粒度的软件实体，例如文件、模块或包进行预测[10,11,12,13,8,14,15]。这些针对粗粒度软件实体的技术在实际应用中遇到了挑战。例如，软件缺陷预测模型可能会预测一个巨大的文件存在缺陷，但是对于开发者来说查看整个文件来寻找缺陷非常耗费精力和时间[16]。同时，这样一个巨大的文件可能被很多开发者修改过，因此寻找一个合适的开发者来对这个文件检查也是一个非常困难的任务[17]。

为了应对上述挑战，软件工程领域的研究者提出即时缺陷预测(Just-in-Time Defect Prediction)技术[17,18,19,20,21,22,23,24,25,26,27,28]。即时缺陷预测技术是指预测开发者每次提交的代码变更(Code Change)是否存在缺陷的技术[19]。在即时缺陷预测技术中，被预测的软件实体是一次代码变更。即时缺陷预测技术具有即时性，具体体现在这种预测技术可以在开发者提交一次代码变更后即对变更代码进行缺陷分析，预测其存在缺陷的可能性。这种技术可以有效应对传统缺陷预测技术面临的挑战，主要体现在以下三方面:

(1) **细粒度**。相比模块或文件级缺陷预测，变更级预测关注更加细粒度的软件实体。开发者可以花费更少的时间和精力去审查被预测为有缺陷的代码变更;

(2) **即时性**。即时缺陷预测技术可以在代码变更提交时进行预测，此时，开发者仍然对变更代码具有鲜活的记忆，不必花费时间来重新理解自己提交的代码变更，有助于更及时的修复缺陷;

(3) **易追溯**。开发者提交的代码变更中保存了开发者的信息，因此项目管理人员可以更便捷的找到引入缺陷的开发者，有助于及时分析缺陷引入原因，帮助完成缺陷分派[19]。

上述优势也吸引了软件工程领域研究者的广泛关注，产生了一批即时缺陷预测研究成果。其中 Mockus 和 Weiss 首次提出了在代码变更层次上进行缺陷预测[29]，他们提出的预测技术中被预测软件实体是由多次代码变更提交组成的代码变更组合。Kim 等人在 2008 年 TSE 中首次提出对每次代码变更进行缺陷预测[17]。Kamei

等人在 2013 年 TSE 中首次将这种缺陷预测技术称作为即时缺陷预测技术[19]。近年来，即时缺陷预测技术由于其细粒度、即时性和可追溯的优势，成为了缺陷预测领域的研究热点。在大量工作中，研究者们针对数据标注、特征提取、模型构建、模型评估等方面提出了大量有价值的理论和技术，同时也产生了一些分歧和讨论，例如 Yang 等人提出使用简单无监督的即时缺陷预测技术，取得了比有监督缺陷预测相当或更好的效果[25]，针对该成果，有研究者针对其建模技术和模型评估等方面展开了讨论，并指出在模型评估中需要综合考虑不同类型的性能指标[27,26]。

此外，即时缺陷预测也吸引了来自工业界的关注。例如 Shihab 等人针对一家大型软件公司 60 个团队展开了实证研究[18]，Kamei 等人在 5 个公司项目上展开了实证研究[19]，展示了即时缺陷预测技术的实用性。由此可见，即时缺陷预测技术已经引起了软件工程学术界和工业界的关注，产生了一批优秀研究成果，同时也面临着缺乏统一建模技术和评估指标的挑战。然而，目前尚没有研究工作对当前该领域的研究进展进行梳理和归纳，鉴于此，本文拟针对当前即时缺陷技术研究进展，从数据标注、特征提取、模型构建和模型评估等方面进行梳理、归纳和总结，并总结了当前该领域存在的主要问题和未来的发展方向。

文献选取方式。本文采用以下流程完成对文献的索引与选取。首先定义文献选取标准：该文献针对即时缺陷预测技术中数据标注、特征提取、模型构建、模型评估提出新理论、新技术，或者该文献对即时缺陷预测相关理论和技术提供实证研究支持。此外，该文献应公开发表在期刊，会议，技术报告或书籍中。依据以上标准，本文通过以下三步骤对文献进行检索和筛选。

1) 本文文献搜索主要通过 ACM Digtal Library、IEEE Xplore Digital Library、Springer Link Online Library 以及 Google Scholar 等。论文检索的关键字包括 just-in-time defect prediction、change level defect prediction、buggy change prediction 等，同时在标题、摘要、关键词和索引中进行检索。
2) 本文对软件工程领域的主要期刊与会议进行在线搜索，具体包括 TOSEM、TSE、EMSE、IST、ICSE、FSE、ASE、ICSME、MSR、SANER 等，搜索时间从 2008 年(即针对每次代码变更预测缺陷的首次提出时间)开始。
3) 本文基于上述步骤所获取文献集合，对文献逐一查看，从文献的参考文献中进一步筛选出与即时缺陷预测理论相关的文献。

通过以上文献索引与选取，共计 34 篇文献纳入本文后续文献总结中。其中 19 篇文献与即时缺陷预测技术直接相关，这些文献为该技术中数据标注、特征提取、模型构建和评估等方面提出了新理论和新技术，另外 15 篇文献对即时缺陷预测相关理论和技术(如特征提取)提供实证研究支持。图 1 展示了本文所总结的文献分布情况，其中包含 TSE 论文 9 篇，MSR 论文 5 篇，FSE 论文 4 篇，ICSE 论文 3 篇，ASE 论文 2 篇，ICSM/ICSME 论文 2 篇，其他期刊和会议论文各 1 篇。

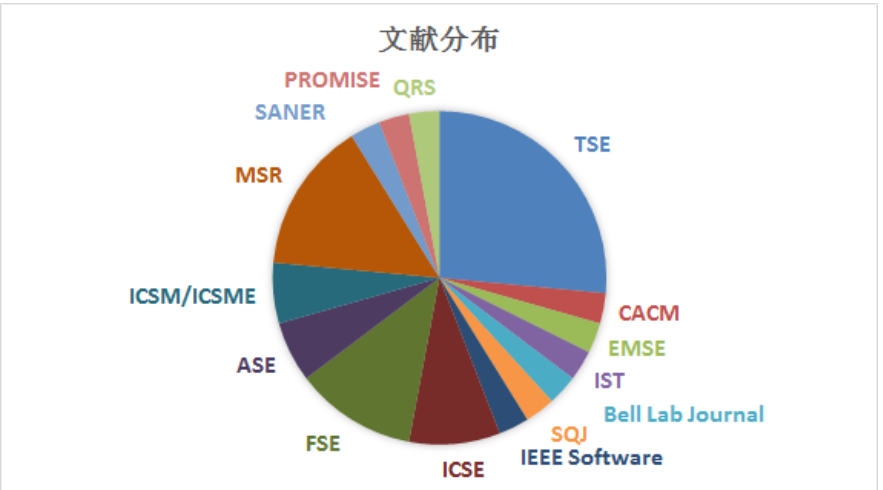

Fig.1 Distribution of surveyed literatures
图 1 文献分布

本文整体结构。本文第二部分介绍即时缺陷预测技术的整体技术框架，并从数据标注、特征提取和模型构建三方面进行归纳和梳理；本文第三部分介绍即时缺陷预测模型评估方法，主要从实验验证方法和技术评价指标两方面进行总结和归纳。本文第四部分总结了即时缺陷预测领域面临的关键问题，包括科学问题、技术难点和工程实现三个方面。本文第五部分介绍即时缺陷预测技术未来发展趋势。本文第六部分是对本文的总结。

2 即时缺陷预测技术

图 2 展示了即时缺陷预测技术的一般过程，主要包括三个阶段，数据标注、特征提取和模型构建。其中数据标注阶段主要依赖于版本控制系统(例如 Git)和缺陷追踪系统(例如 Bugzlla 或 Jira)，将代码变更标注为缺陷变更(Buggy)或非缺陷变更(Clean)；特征提取阶段主要通过提取不同维度的特征来表示代码变更；模型构建阶段主要依赖于机器学习技术构建预测模型，当新的代码变更提交时，模型将预测其缺陷可能性。本章将对上这三个阶段进行详细描述。

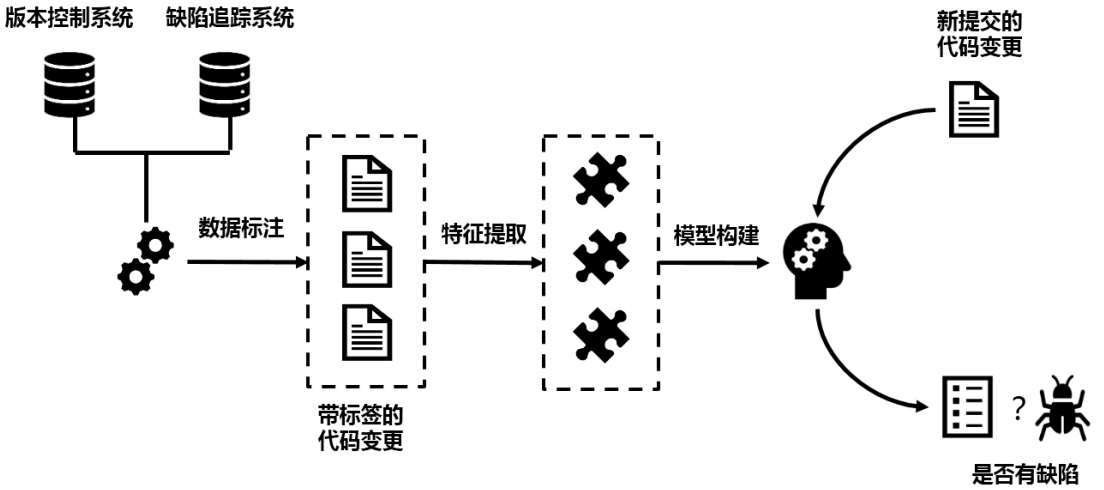

Fig.2 Research framework of Just-in-time defect prediction
图 2 即时缺陷预测技术一般框架

2.1 数据标注

即时缺陷预测技术中的数据标注是指将代码变更标注为缺陷引入变更和非缺陷引入变更。准确的数据标注是完成模型训练和模型评估的前提。

为了实现数据标注，需要利用软件项目的历史变更数据，识别缺陷引入变更。识别缺陷引入变更是一个复杂问题，需要考虑代码的动态演进和代码变更之间的关联关系。Sliwerski, Zimmermann 和 Zeller 在 2005 年首次提出用于自动识别引入缺陷的代码变更方法，并以作者名字的首字母命名该算法，即 SZZ 算法[30]。

随后，SZZ 算法成为用于从软件项目代码仓库和缺陷仓库中识别缺陷引入变更的通用框架[30]。为即时缺陷预测技术命名的研究者 Kamei 和 Shihab 评价 SZZ 算法是修改了缺陷预测研究领域的游戏规则 (Game Changer)，促进了即时缺陷预测技术诞生和发展[31]。

SZZ 算法的一般框架包含了 4 个步骤，图 2 使用 Apache 项目 ActiveMQ 中的一个缺陷(AMQ-1381)作为一个具体的例子来详细描述这个算法框架的 4 个步骤。

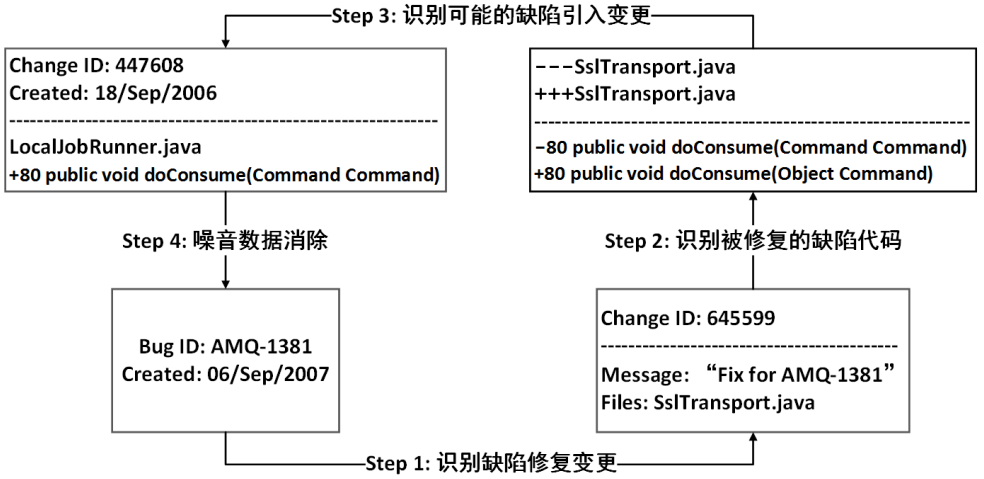

Fig.3 SZZ framework

图 3 SZZ 算法一般框架

(1) **识别缺陷修复变更**。扫描存储在版本控制系统的所有历史数据，即所有代码变更，识别日志中包含缺陷 ID 的代码变更。这些代码变更被识别为缺陷修复变更。如图 2 所示，识别修复 AMQ-1381 的代码变更 ID 为 645599。

(2) **识别被修复的缺陷代码**。利用版本控制系统实现的 diff 算法来识别上述修复缺陷的代码变更修改的代码行。这些被修改的代码行被识别为缺陷代码。如图 2 所示，被代码变更#645599 修改的代码中包含一个函数声明，其中 Command 参数类型是错误的，它的正确类型是 Object 而不是 Command。

(3) **识别可能的缺陷引入变更。**使用代码版本控制系统中的 annotate 命令回溯代码变更提交历史，第一次引入缺陷代码的变更被识别为可能的缺陷引入变更。如图 2 所示，代码变更#447068 引入了在第二步中发现的错误函数声明。

(4) **噪音数据消除**。从可能的缺陷引入变更中去除可能存在的噪音数据。噪音数据是指被误标记为引入缺陷而实际未引入缺陷的变更(False Positive)。Sliwerski 等人提出缺陷引入变更应当在该缺陷被报告之前被提交[30]。因此，提交时间晚于缺陷报告时间的代码变更在 SZZ 实现中会被当作噪音去除。图 2 中所示代码变更#447608 的提交时间在缺陷报告创建时间之前，因此，这个代码变更最终被确认是引入缺陷 AMQ-1381 的代码变更。

由于 SZZ 算法在众多软件工程研究工作中的基础性地位，研究者开始关注 SZZ 产生数据的质量[32,33,34]。在这些工作中，研究者们发现原始 SZZ 产生的数据存在噪音，进一步，他们对 SZZ 的噪音数据进行处理，提出了不同类型的改进版 SZZ 实现。为了描述方便，本文将 Sliwerski 等人提出的基础 SZZ 实现称作 Basic SZZ，简称 B-SZZ。

本节对研究者们在 B-SZZ 基础上做出不同版本的改进 SZZ 进行简要概述。主要包括以下三种：

Annotation Graph SZZ (AG-SZZ)。Kim 等人注意到 B-SZZ 在第二步和第三步中都引入了噪音数据。他们发现，B-SZZ 在执行第二步时会错误地把空行、注释行以及代码风格的修改识别为用于缺陷修复代码。并且，B-SZZ 在执行第三步时错误地把对代码风格的修改识别为引入缺陷的代码修改。而事实上，这些代码修改不会对软件行为造成任何影响。为了避免这些噪音，Kim 等人提出 SZZ 算法在第二步中忽略对空行、

注释行以及代码风格的修改，并且在第三步中使用 Annotation Graph 来对代码变更提交历史进行追踪，识别可能的缺陷引入变更[32]。其中，Annotation Graph 是一种对软件代码演进过程的追踪工具[35]，其基本思路在于利用版本控制系统的 diff 算法，追踪每次代码变更中对代码的增加、删除以及修改。因此，Annotation Graph 可以帮助 SZZ 在第三步中发现对代码风格的修改，忽略这种修改继续追踪代码提交历史，识别真正引入缺陷的代码变更。

Table 1. Summary of different SZZ implementations

表 1. 不同 SZZ 实现算法简介

标注方法	描述	消除噪音类型	相关文献
Basic SZZ（B-SZZ）	第一种 SZZ 算法实现；在识别引入缺陷步骤中使用版本控制系统的 annotate 命令追踪变更提交历史	避免创建时间在缺陷报告之后的变更被误标记为引入缺陷 (包括 False Positive)	Sliwerski et al.[30]
Annotation Graph SZZ（AG-SZZ）	以 B-SZZ 为基础；在识别缺陷修复代码步骤中忽略对空行、注释行和代码风格的修改；在识别引入缺陷变更步骤中忽略修改代码风格的变更，并使用 Annotation Graph 追踪变更提交历史	1）避免对空行、注释行和代码风格的修改被误识别为缺陷修复代码，避免引入这些代码的变更被误标记为引入缺陷 (包括 False Positive)； 2）避免修改代码风格变更被误标记为引入缺陷同时遗漏真正引入缺陷的变更（包括 False Positive 和 False Negative）	Kim et al.[32]
Meta-change Aware SZZ (MA-SZZ)	以 AG-SZZ 为基础；在识别引入缺陷变更步骤中忽略包括分支创建、合并和修改文件属性的变更（即元变更）	避免元变更被误标记为引入缺陷同时遗漏真正引入缺陷的变更 (包括 False Positive 和 False Negative)	Da Costa et al.[33]
Refactoring Aware SZZ (RA-SZZ)	以 MA-SZZ 为基础；整合重构代码检测工具，在识别缺陷修复代码和识别引入缺陷变更两步骤中忽略代码重构修改	1)避免将代码重构修改被误识别为修复缺陷代码，避免引入被重构代码的变更被误识别为引入缺陷；（包括 False Positive） 2）避免重构类变更被误标记为引入缺陷同时遗漏真正引入缺陷的变更（包括 False Positive 和 False Negative）	Neto et al.[34]

Meta-change Aware SZZ (MA-SZZ)。Da Costa 等人发现 AG-SZZ 产生的数据中仍然存在噪音，他们发现 AG-SZZ 会把创建分支、合并分支以及修改文件属性的代码变更识别为缺陷引入变更[33]。Da Costa 等人将创建分支、合并分支以及修改文件属性的代码变更称为元变更(Meta-change)。事实上，元变更不会对软件行为造成影响。这种噪音的产生是由于 AG-SZZ 在第三步追溯代码提交历史的过程中，遇到元变更会被阻断，从而将元变更识别为引入缺陷的代码变更。为了去除这些噪音，Da Costa 等人在 AG-SZZ 的基础上进行改进，提出元变更感知 SZZ (Meta-change Aware SZZ)，简称 MA-SZZ。MA-SZZ 在第三步追踪代码变更提交历史的过程中会忽略元变更，识别真正引入缺陷的代码变更。

Refactoring Aware SZZ (RA-SZZ)。Kim 等人指出 SZZ 可能会将重构代码变更识别为引入缺陷的代码变更[32]。重构代码是一种既不会更改软件的外部行为也不会提升软件内部结构的代码修改过程[36]。Neto 等人对重构代码对 SZZ 算法的影响进行了分析[34]。作者指出，SZZ 在第二步时会错误地把代码重构识别为用于修复缺陷的代码修改，并且在第三步中会错误地把代码重构识别为引入缺陷的代码修改。为了处理这些噪音，Neto 等人在 MA-SZZ 的基础上，提出重构感知 SZZ (Refactoring Aware SZZ)，简称 RA-SZZ。在 RA-SZZ 中，Neto 等人通过将 RefDiff 工具集成到 SZZ 中实现对重构代码进行检测。RefDiff 是一种用于检测 Java 代码中代码重构修改的工具[37]。RA-SZZ 在执行第二步时会忽略由 RefDiff 检测出的代码重构修改，并且在第三步追踪代码变更提交历史过程中，忽略由 RefDiff 检测出的代码重构修改，识别真正引入缺陷的代码变更。然而，RefDiff 在实际使用中存在局限性，因此限制了 RA-SZZ 的实际应用。比如 RefDiff 只能用于检测 Java 代码中的代码重构，因此对 RA-SZZ 造成了程序语言的限制。

表 1 展示了现有工作中研究者们提出的不同数据标注方法，以及它们消除的噪音类型。表中 False

Positive、False Negative 分别指被误标记而实际未引入缺陷的变更、未被识别出的真正引入缺陷的变更。

2.2 特征提取

在即时缺陷预测技术中，特征提取是指利用代码变更原始数据计算特征来表征变更的过程。该步骤是即时缺陷预测技术的关键步骤。本节分为两小节对特征提取进行描述。第一小节按照时间顺序概括即时缺陷预测研究中特征提取技术的研究进展。第二小节对现有工作提出的特征归纳总结，并将所有特征划分为 9 个维度进行介绍。

2.2.1 特征提取研究进展

即时缺陷预测研究者先后提出不同类型的变更特征，包括基于变更元数据的特征、基于变更代码内容的特征、基于软件演进过程的特征和基于多源软件制品的特征。本小节按照时间先后顺序分别对这四种特征进行概述。

基于变更元数据的特征。变更元数据是指描述变更属性(例如开发者、提交时间、变更日志、修改文件、每个文件增加和减少的代码行数等)的数据。这些元数据可以直接通过软件项目变更提交记录获取。基于变更元数据，Mockus 和 Wiss 提出变更增加减少行数、修改文件数量、开发者经验、变更类型等特征[29]。这些特征具有坚实的实证研究基础[38,39,40,41]。它们容易理解且计算简单。由于代码变更提交记录可以直接通过软件项目的版本控制系统获取，这些特征可以应用到大量软件项目中，具有普遍性。在后续工作中，这些特征得到了广泛的使用[17,18,19,20,21,22,23,24,25,26,27,28]。

基于变更代码内容的特征。基于变更元数据的特征对于表征变更是不完整的。这些特征并没有考虑变更具体修改的代码内容。因此，研究者们提出了基于变更代码内容的特征。Kim 等人基于实证研究成果提出使用代码复杂度特征来表征变更[42,43,17]。同时，他们还使用代码变更代码、日志和文件名的词频率特征来量化代码变更内容。Jiang 等人提出使用变更提交前后相关代码文件的抽象语法树相同类型节点数量的差值表征变更[20]。这些特征依赖变更具体的代码内容。研究者们将这些基于变更代码内容的特征与基于变更元数据的特征结合，构建预测模型[17,20]。

相比基于变更元数据的特征，基于变更代码内容的特征数量明显增加。比如，变更词频率特征有成千上万个[21]。特征过多容易造成著名的维度灾难问题[44]。Shivaji 等人指出特征过多对即时缺陷预测模型产生负面影响，使用特征选择减少特征数量可以显著提升预测模型的性能[21]。因此，在使用基于变更代码内容的特征时，需要进行特征选择。

在计算基于变更代码内容的特征时，需要先提取变更相关文件在变更提交前后的代码内容，然后对变更前后的文件内容分别计算复杂度特征和词频率特征及构建抽象语法树[17,20]。相比基于变更元数据的特征，这些特征计算难度较大而且比较费时。对于不同编程语言，代码复杂度计算以及抽象语法树构建存在差异，使这些特征的实用性受到影响。

基于软件演进过程的特征。在后续即时缺陷预测工作中，研究者们受传统缺陷预测研究的启发，提出了基于软件演进过程的变更特征，即基于项目代码修改历史量化变更。

传统缺陷预测研究表明，基于软件演进过程的特征(例如文件修改次数)比基于代码内容的静态特征对于预测缺陷更加有效[45]。Kamei 等人提出从文件修改历史角度提取变更特征，例如变更相关文件被修改的次数、修改变更相关文件的开发者人数等[19]。相比基于变更代码内容的特征，这些特征数量显著减少，避免了维度灾难问题和模型构建的大量开销。同时，Kamei 等人提出的基于文件修改历史的特征可以直接通过变更提交历史记录计算。因此，这些特征具备普遍性。在后续即时缺陷预测工作中，这些特征被广泛应用[22,24,25,26,27,28]。

基于多源软件制品的特征。上述三种变更特征的提取都是基于软件项目的代码版本控制系统。近年来，软件工程研究者发现即时缺陷预测与多源软件制品相关。例如变更提交过程中，除版本控制系统外，开发者会在软件项目多个管理系统对变更进行审查、测试等操作[46]。因此，有研究者提出结合多源软件制品提取多维代码变更特征来增强即时缺陷预测性能，例如 McIntosh 和 Kamei 首次从项目的代码审查系统提取特

征来表征代码变更[28]。由于现代软件项目中对代码审查的广泛使用，这些基于代码审查系统的特征具备普遍性，在后续工作中，这些特征可能会得到更多的应用。

2.2.2 特征类型

现有工作中，研究者提取了大量特征用于表征代码变更。本小节在第一小节基础上对现有即时缺陷预测研究提出的变更特征进一步归纳总结，第一小节的四种类型变更特征被进一步分为 9 个维度，包括规模(Size)、代码分布(Diffusion)、目标(Purpose)、开发者经验(Developer Experience)、复杂度(Complexity)、文本(Text)、代码结构(Structure)、文件修改历史(File History)和代码审查(Code Review)。

基于变更元数据的特征包括规模、代码分布、目标和开发者经验等特征维度。表 2 中归纳了基于变更元数据的特征以及它们的使用动机。

Table 2. Summary of features based on meta-data of changes

表 2. 基于变更元数据的特征简介

特征维度	特征名	描述	动机	相关文献
规模	**LA/LD**	变更增加、减少代码行数(added lines and deleted lines)	变更增加和减少代码行越多，越有可能引入缺陷	Mockus and Weiss[29]
	CA/CD	变更增加、减少代码段数(added chunks and deleted chunks)	变更增加、减少代码段越多，对软件代码的影响越大，越有可能引入缺陷	Shihab et al.[18]
	LT	变更相关文件在变更提交前的代码行数(lines of code of files touched by the change)	文件越大，对该文件的修改越有可能引入缺陷	Mockus and Weiss[29]
代码分布	**NS**	变更修改的子系统数量(number of subsystems)	修改子系统越多的变更越有可能引入缺陷	Mockus and Weiss[29]
	ND	变更修改的代码目录数量(number of directories)	修改代码目录越多的变更越有可能引入缺陷	Kamei et al.[19]
	NF	变更修改的文件数量(number of files)	修改文件数量越多的变更越有可能引入缺陷	Mockus and Weiss[29]
	Entropy	修改代码在变更相关文件中的分布（用信息熵计算）	信息熵越大，变更代码在相关文件中越分散，开发者需要理解更多代码，越有可能引入缺陷。	Hasssan[11]
目标	**FIX**	该变更是否修复缺陷	修复缺陷的变更更加复杂，更容易引入缺陷	Mokus and Weiss[29]
	NBR	与该变更相关的缺陷报告数量(number of bug reports)	相关的缺陷报告越多，变更需要修改更多代码，更有可能引入缺陷	Shihab et al.[18]
经验	**EXP**	开发者已提交变更数量(experience)	有更多经验的开发者不容易引入缺陷	Mockus and Weiss[29]
	REXP	开发者近期提交变更数量(recent experience)	近期经常修改代码的开发者对项目开发更加熟悉，不容易引入缺陷	Mockus and Weiss[29]
	SEXP	开发者已提交变更中影响到该变更相关子系统的数量(subsystem experience)	开发者对熟悉的子系统进行修改，不容易引入缺陷	Mockus and Weiss[29]
	Awareness	开发者已提交变更中,影响到相关子系统的变更占所有影响这些子系统变更的比例	对子系统修改越多，开发者对子系统越熟悉，越不容易引入缺陷	McIntosh and Kamei[28]

规模维度。该维度用来表征变更修改代码的规模。Moser 等人观察到变更修改代码规模越大，越有可能引入缺陷[45]。现有工作中，研究者们都使用变更增加、减少代码行数来量化变更规模[29,17,18,19,20,21,22,23,24,28]。此外，研究者们使用其他粒度对变更代码规模进行量化。Shihab 等人提出使用变更增加和减少代码段（chunk）数来量化变更规模[18]。Kamei 等人提出使用变更提交前变更相关文件代码行数量化变更规模[19]。同时，Kamei 等人发现变更增加行数和减少代码行数是高度相关的。为了避免这种相关性对预测模型造成

影响，Kamei 等人提出使用相对增加行数和相对减少行数量化变更规模。相对增加行数、相对减少行数分别指变更实际增加、减少行数与变更提交前其相关文件代码行数的比值[47]。

代码分布维度。该维度用来表征变更修改代码在相关文件中的分布。研究表明，对于修改代码分布在多个文件的变更，其开发者需要理解更多代码，因此这种更加分散的代码变更有可能引入缺陷[48]。Kamei 等人提出使用变更修改的文件数量、文件夹数量和子系统数量等特征来量化变更代码分布[19]。Hassan 提出使用变更代码分布的信息熵(Entropy)预测缺陷，并且验证了该特征的有效性[11]。因此，Kamei 等人提出使用该特征对有缺陷变更进行预测[19]。

目标维度。该维度用来表征提交代码变更的目标。开发者提交变更的目标包括修复缺陷、实现新功能、重构、增加文档等[49]。研究表明，修改缺陷的变更相比其他类型变更更复杂，更有可能引入缺陷[30]。因此，在大量即时缺陷预测工作中，研究者利用变更是否修复缺陷作为特征[18,19,22,24,28]。此外，Shihab 等人还提出使用变更相关缺陷报告的数量来量化代码变更目标[18]。

开发者经验维度。该维度用来量化代码变更的开发者经验。研究表明，开发者经验会影响软件质量[50]。在即时缺陷预测工作中，研究者都利用变更开发者已提交变更数量量化开发者经验[17,18,19,20,21,22,23,24,28]。此外，Kamei 等人提出使用开发者近期提交变更的数量以及变更提交前，开发者已提交变更中影响该变更相关子系统的变更数量量化开发者经验[19]。McIntosh 和 Kamei 提出使用变更提交前，开发者已提交变更中影响该变更相关子系统的变更占版本控制系统中所有影响这些子系统变更的比例，来量化开发者对这些子系统的开发经验[28]。

基于变更代码内容的特征包括复杂度、文本和代码结构等特征维度。表 3 中归纳了基于变更代码内容的特征以及它们的使用动机。

Table 3. Summary of features based on changed code

表 3. 基于变更代码内容的特征简介

特征维度	特征名	描述	动机	相关文献
复杂度	Complexity	变更相关文件在变更提交前后复杂度指标的差值，复杂度指标如代码总行数(LOC)、注释代码行数、圈复杂度等	变更代码越复杂，越有可能引入缺陷	Kim et al.[17]
文本	LogTF	变更日志的词频率特征	变更日志的内容与变更是否引入缺陷相关	Kim et al.[17]
	FileTF	变更相关文件内容在变更提交前后词频率特征的差值	变更修改代码内容与变更是否引入缺陷相关	Kim et al.[17]
	FilenameTF	变更修改目录名和文件名的词频率特征	目录名和文件名反映代码模块信息和语义信息，这些信息与变更是否引入缺陷相关	Kim et al.[17]
代码结构	Structure	变更相关文件在变更提交前后抽象语法树的差异	变更引起的抽象语法变化可能与变更是否引入缺陷相关	Jiang et al.[20]

复杂度维度。该维度用来表征变更修改代码的复杂度。在传统缺陷预测工作中，研究者们通常使用文件或者模块代码的复杂度来预测文件或者模块中是否存在缺陷[43,51]。研究者发现越复杂的代码越容易引入缺陷[52]。Kim 等人将描述文件代码复杂度的特征引入到即时缺陷预测技术中[17]。为量化变更引起的代码复杂度变化，作者对变更提交前后该变更相关文件的代码内容分别计算了 61 个复杂度指标，计算对应复杂度指标的差值作为特征对有缺陷变更进行预测。

文本维度。该维度利用词袋(Bag of words)模型从变更修改代码、日志和文件名中提取词频率特征。词袋模型是一种将文本表示为词频率向量的方法[53]。Kim 等人将变更文本特征引入到即时缺陷预测技术中[17]。对于变更日志和相关目录名、文件名，作者利用词袋模型将它们转换为词频率向量。为了量化变更引

起的代码内容变化，作者对变更前后该变更相关文件的代码内容分别计算词频率特征，计算相应词频率的差值，并且使用这些差值作为变更特征。

代码结构维度。该维度用于量化代码变更前后代码结构的差异。即时缺陷预测研究者通过抽象语法树(Abstract Syntax Tree，AST)来量化代码的结构，从而计算代码变更前后的代码结构差异[20,23]。抽象语法树是一种表示源代码抽象结构的工具[54]。在抽象语法树中，源代码中的条件语句、循环语句和函数等表示为不同类型的节点。Jiang 等人提出使用在变更提交前后，该变更相关文件抽象语法树中相同类型节点的数量差量化变更引起的代码结构变化[20]。

基于软件演进过程的特征包括文本修改历史维度。表 4 中归纳了基于软件演进过程的特征以及它们的使用动机。

Table 4. Summary of features based on software evolution process

表 4. 基于软件演进过程的特征简介

特征维度	特征名	描述	动机	相关文献
文件修改历史	**NDEV**	对该变更相关文件进行过修改的开发者数量(number of developers)	对文件进行过修改的开发者越多，修改该文件的变更越可能引入缺陷	Shihab et al.[18]
	NUC	对该变更相关文件进行过修改的变更数量(number of unique changes)	文件修改次数越多，开发者对该文件修改需要理解更多代码，变更修改该文件越可能引入缺陷	Shihab et al.[18]
	NFIX	对该变更相关文件进行过修改的缺陷修复变更数量(number of fixes)	文件已知的缺陷越多，修改该文件的变更越容易引入缺陷	Shihab et al.[18]
	AGE	修改过该变更相关文件的最近变更与该变更时间差平均值	近期提交的变更更容易引入缺陷	Kamei et al.[19]

Table 5. Summary of features based on different artifacts of multiple repositories

表 5. 基于多源软件制品的特征简介

特征维度	特征名	描述	动机	相关文献
代码审查	**Iterations**	变更被合并到项目代码仓库之前被修正的次数	在合并到项目代码仓库之前，变更修正多次减小其引入缺陷的可能性	McIntosh and Kamei[28]
	Reviewers	对该变更进行审查的人数	代码审查人员数量越多，审查人员可以从变更中找到更多问题，减小变更引入缺陷的可能性	McIntosh and Kamei[28]
	Comments	该变更审查意见数量	开发者对变更讨论越少，该变更越有可能引入缺陷	McIntosh and Kamei[28]
	Review Window	该变更代码审查时间，即变更被创建与变更被合并到项目代码仓库的时间差	开发者对变更审查时间越短，该变更越有可能引入缺陷	McIntosh and Kamei[28]

文件修改历史维度。该维度用来量化变更相关文件的修改历史。实证研究表明，文件修改历史越复杂(例如被多次修改、被多个开发者修改等)，其越有可能存在缺陷[45,55,56]。鉴于此，即时缺陷预测研究者提出使用变更提交前变更相关文件的修改次数、修改这些文件的开发者人数作为特征量化文件修改历史对有缺陷变更进行预测[18,19,22,24,28]。此外，Shihab 等人提出使用修改变更相关文件的历史变更中缺陷修复变更的数量量化变更相关文件的修改历史[18]。Kamei 等人提出使用变更提交前修改该变更相关文件的最近变更与该变更的时间差作为特征量化变更文件的修改历史[19]。

基于多源软件制品的特征包括代码审查维度。表 5 中归纳了基于多源软件制品的特征以及它们的使用动机。

代码审查维度。该维度用来量化变更被合并到项目代码仓库前的代码审查过程。近年来，现代代码审查被广泛应用到开源与商业项目中。实证研究表明，代码审查质量与软件缺陷相关[57]。因此，McIntosh

和 Kamei 将代码审查过程的特征引入到即时缺陷预测研究中。作者提出使用变更审查过程中变更被重复修正的次数、审查人数、审查意见数量以及审查时间等特征来量化对变更的代码审查质量[28]。

2.3 模型构建

现有即时缺陷预测技术模型构建方法主要包括两种类型：有监督(Supervised)建模技术和无监督(Unsupervised)建模技术[27]。本章将分别对有监督和无监督即时缺陷预测两种建模技术进行介绍。

2.3.1 有监督建模

有监督即时缺陷预测是指利用有监督机器学习方法，从已知标签(标签即是否存在缺陷)的代码变更数据中学习，从而构建预测模型的建模技术。现有工作中，大多数即时缺陷预测工作使用了有监督建模技术[17,18,19,20,22,23,24,28]。其一般过程包括:

(1) 数据标注。利用数据标注技术，将历史代码变更标记为有缺陷或无缺陷;

(2) 特征提取。从代码变更的不同角度提取特征，并用特征向量来表示每个代码变更;

(3) 模型训练。利用有监督机器学习技术，从标注后的代码变更数据中学习，构建分类或回归模型;

(4) 模型应用。当有新的代码变更提交时，应用预测模型预测该代码变更存在缺陷的可能性。

此外，根据训练数据的来源不同，有监督即时缺陷预测又可以分为同项目(Within-Project)即时缺陷预测和跨项目(Cross-Project)即时缺陷预测两种建模技术。

同项目即时缺陷预测。同项目即时缺陷预测是指训练数据与测试数据来自同一个项目，即使用同项目的历史代码变更数据进行训练，利用有监督学习技术构建预测模型[17,18,19,20,21,23,28,26,27]。要使预测模型取得良好的性能，其前提条件是预测目标项目中拥有足够多的历史数据。现代大型软件项目中都存在大量变更数据，为同项目即时缺陷预测建模技术提供了数据基础。而同一个项目中引入缺陷的代码修改具备相似性，使用预测目标项目的历史数据来构建预测模型可以取得较高的性能。因此，大量即时缺陷技术的应用研究都采用了同项目即时缺陷预测技术[18,19,20,21,23,28]。

然而，同项目即时缺陷预测技术的缺点在于，如果预测目标项目是新项目，这样的项目无法为同项目即时缺陷预测技术提供足量已知标签的历史数据用于训练模型[22,24]。

跨项目即时缺陷预测。为应对项目内的即时缺陷预测技术在对新项目预测时缺乏训练数据的问题，软件工程领域的研究者提出跨项目即时缺陷预测建模技术[22,24]。跨项目即时缺陷预测建模技术是指训练数据与测试数据来自不同项目，即使用其他项目的历史变更数据训练预测模型，利用该模型在目标项目上完成预测[22,24]。

Fukushima 等人首次提出并研究了跨项目即时缺陷预测技术[22]。作者在后续工作中对这一技术进行了进一步分析[24]。这些分析研究表明，相比同项目即时缺陷预测技术，跨项目即时缺陷预测技术在大多数情况下存在着性能劣势。为了应对跨项目即时缺陷预测技术的性能劣势，作者提出了三种方法可以有效提高跨项目即时缺陷预测技术的性能：1) 使用相似的软件项目变更数据作为训练集；2) 使用多个项目融合后的变更数据作为训练集；3) 使用多个项目的变更数据分别作为训练集训练模型，在预测时以投票方式获得预测结果。这些研究成果为实践中跨项目即时缺陷预测的应用提供了指导。

2.3.2 无监督建模

有监督即时缺陷预测技术的缺点在于需要消耗大量的资源，因为有监督建模技术需要利用大量已知标签的训练数据来训练预测模型[25]。为应对这一挑战，研究者提出无监督即时缺陷预测技术[25]。

无监督即时缺陷预测是指利用未知标签变更数据构建预测模型的技术。相比有监督即时缺陷预测建模技术，无监督即时缺陷预测建模技术的优点在于更符合实际使用过程的需要。无监督的预测模型在构建时，不需要使用有标签的数据，因此节省了数据标注的开销。其一般过程包括：

(1) 特征提取。从代码变更的不同角度提取特征，并用特征向量来表示每个代码变更;
(2) 模型构建。基于代码变更特征构建无监督模型。目前，在即时缺陷预测领域，主要是基于变更特征排序的方法，这种方法通过对新提交的变更排序，对开发者审查任务进行调度，使开发者在审查一定量代码情况下找到更多缺陷，即工作量感知的即时缺陷预测。

Yang 等人首次提出无监督即时缺陷预测建模技术[25]。实证研究表明，代码存在缺陷数量与代码规模具有对数关系[16]。因此，相比规模巨大的代码变更，审查小变更时发现缺陷所需工作量更少。Yang 等人基于该发现，提出基于变更特征排序的方法，使开发者按照该排序审查变更代码时，在一定审查工作量下找到更多缺陷。比如，使用变更修改文件数量作为排序的特征，将文件数量少的变更排在更高的优先级。Yang 等人对 12 个变更特征分别建立类似的无监督模型，然后在 6 个项目数据上进行验证后发现，部分无监督模型相比有监督即时缺陷预测模型能够对开发者代码审查工作进行更合理的调度。同样检查占总共修改代码行数 20%的代码，相比有监督即时缺陷预测模型，无监督即时缺陷预测模型能够使开发者检查到更多缺陷引入变更。

Yang 等人提出的无监督即时缺陷预测技术引起了相关研究者的关注，Huang 等人和 Fu 等人分别对 Yang 等人的无监督即时缺陷预测建模技术做了进一步的分析[26,27]。这些研究通过将无监督和有监督建模技术的分析，揭示了无监督即时缺陷预测建模技术的缺点，主要体现在以下几个方面.首先，同样检查 20%的代码，无监督缺陷预测模型向开发者推荐的可能引入缺陷的变更数量远远超过有监督即时缺陷预测模型，而检查更多代码变更需要消耗更多人力、物力[26]。其次，无监督缺陷预测模型误报率比有监督缺陷预测模型高很多[26]。Fu 等人指出先使用有标签的训练数据选择合适的特征，然后实施 Yang 等人的无监督方法，能达到更好的效果[27]。

尽管 Yang 等人提出的无监督即时缺陷预测建模技术有上述缺陷，他们提出的对于有监督即时缺陷预测建模技术所面临的挑战是需要研究者重视的，并且他们提出无监督即时缺陷预测建模技术可以作为可选解决方案在实际中运用。

3 即时缺陷预测评估方法

在即时缺陷预测的工作中，为了验证和对比模型效果，研究者需要对提出的缺陷预测模型进行实验验证，并通过不同的评价指标对模型性能进行评估。本章将详细介绍在即时缺陷预测研究中使用的实验验证方法和性能指标。

3.1 实验验证方法

实验验证方法是指在模型评估中如何将数据集划分为训练集和测试集。本节详细描述当前即时缺陷预测研究中的常用的两种实验验证方法：交叉验证(Cross Validation)和对时间感知(time-wise)的验证方法。

3.1.1 交叉验证

交叉验证是机器学习领域常用的模型验证方法[58]。大部分的即时缺陷预测的工作都采用了 10 折交叉验证的方法[17,18,19,20,21,22,24]。此外，Yang 等人使用了 10 次 10 折交叉验证方法验证缺陷预测模型的性能[25]。在 10 折交叉验证中，首先需要随机打乱数据集，然后将数据集分为 10 等份，即 10 折，之后对每 1 折都作为测试集，另外 9 折作为训练集，训练预测模型并进行验证计算性能指标，一共运行 10 次实验，最后将每次得到的性能指标求平均值作为 10 折交叉验证的评估结果。而在 10 次 10 折交叉验证中，则是将上述 10 折交叉验证运行 10 次，一共运行 10×10 次实验，将每次得到的性能指标求平均值作为 10 次 10 折交叉验证的评估结果。

3.1.2 时间感知验证

近年来，研究者对即时缺陷预测的验证方法有了更多的关注与争论[23,28]。Tan 等人指出交叉验证方法对于即时缺陷预测技术是不适合的，因为变更数据产生具有时间顺序，在交叉验证中随机打乱数据集进行数据划分，会导致预测模型使用未来代码变更信息来预测过去代码变更的标签，从而高估预测模型的性能

[23]。McInstosh 等人发现引入缺陷代码变更的性质会随着时间改变[28]。这种改变会影响即时缺陷预测模型的性能，比如使用 1 年前的变更数据构建预测模型会显著降低模型的性能。McIntosh 等人强调应当使用近期产生的数据来对新变更进行预测。

Yang 等人针对即时缺陷预测提出了一种时间感知的验证方法[25]。这种验证方法首先对数据集按照创建的顺序来排序，然后对数据集分组，在一个月内创建的数据被分为 1 组。在这种验证方法中，训练模型所用的数据集和测试所用的数据都是连续两个月内创建的变更数据，但是训练集和测试集之间则间隔了两个月。比如使用 1 月和 2 月创建的代码变更来预测 5 月和 6 月创建的代码变更。这样验证有以下四点考虑：1) 大多数项目的开发周期是 6-8 星期；2) 每个训练集和测试集之间都会有两个月的时间差，这样训练集中的缺陷在这两个月中会被更多地被开发者发现；3) 使用连续两个月创建的数据保证训练集中有足够多的数据用于训练模型；4) 允许在一个项目中运行多次实验保证计算出的结果的可靠性。Huang 等人和 Fu 等人也都采用了上述这种对时间感知的验证方法[26,27]。

3.2 评价指标

针对模型预测结果，研究者提出不同的指标来量化模型预测效果。本节详细阐述在即时缺陷预测研究中常用的性能指标，包括机器学习领域常用的查准率(Precision)、查全率(Recall)、F1-measure、正确率(Accuracy)、AUC 以及软件工程领域的研究者提出的工作量感知(Effort-aware)指标。

3.2.1 查准率、查全率、正确率和 F1-measure

查准率、查全率、F1-measure 和正确率是机器学习领域常用的评估预测模型的性能指标[58]。大量的即时缺陷预测研究都使用了这些性能指标[17,18,19,20,21,23]。其具体计算方法如下。

预测模型对于一个代码变更的预测结果有四种可能性：1) 将一个有缺陷的代码变更预测为有缺陷(True positive, TP)；2)将一个有缺陷的代码变更预测为没有缺陷(False Positive, FP)；3) 将一个没有缺陷的代码变更预测为没有缺陷(True Negative, TN)；4)将一个没有缺陷的代码变更预测为有缺陷(FalseNegative, FN)。根据预测模型在测试集中这四种预测结果的数量，即可对精确率、召回率、正确率和 F1-measure 进行计算。由于即时缺陷预测研究更加关注有缺陷变更的预测效果，因此本文所描述的查准率、查全率和 F1-measure 都是针对有缺陷的代码变更。

查准率(Precision)是指所有被正确分类的有缺陷变更占所有被分类为有缺陷变更的比例：

$$\text{Precision} = \frac{\text{TP}}{\text{TP+ FP}} \tag{1}$$

查全率(Recall)是指所有被正确分类的有缺陷变更占所有真正有缺陷变更的比例：

$$\text{Recall} = \frac{\text{TP}}{\text{TP+ FN}} \tag{2}$$

F1-measure 是结合了精确率和召回率的综合性性能指标，它是精确率和召回率的调和平均：

$$\text{F1} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision+ Recall}} \tag{3}$$

正确率(Accuracy)是指被正确分类的代码变更占所有代码变更的比例：

$$\text{Accuracy} = \frac{\text{TP+ TN}}{\text{TP+ FP+ TN+ FN}} \tag{4}$$

由于在即时缺陷预测的数据中，没有缺陷的代码变更数量远大于有缺陷的代码变更的数量，预测模型的正确率可能非常高，但这并不意味着预测模型性能好。在评估缺陷预测模型时，即时缺陷预测技术的研究者都会综合考虑查准率、查全率、F1-measure 和正确率的结果。

3.2.2 AUC

在即时缺陷预测研究中，AUC 也是常用的性能指标[19,21,22,24,28]。AUC 即 Area Under the Curve of receiver operating characteristic，指受试者工作曲线(Receiver Operating Characteristic curve，ROC)下的面积。 ROC

曲线是 TP 的比率(true positive rate，TPR)在所有的阈值(threshold)上以 FP 的比率(false positive rate，FPR)为变量的函数曲线。预测模型需要使用一个阈值来对代码变更的标签进行判断。阈值的取值范围是 0 到 1。在预测模型对一个代码变更进行预测时，该模型会对代码变更包含缺陷计算概率值。为了得到预测结果(代码变更有缺陷或者没有缺陷)，该模型对概率值与该阈值进行比较，如果概率值大于阈值，模型将代码变更预测为有缺陷；反之，则将代码变更预测为没有缺陷。这样，就可以算出 TP、FP、TN 和 FN 的数量。因此，查准率、查全率、F1-measure 以及正确率的计算都依赖于预测模型的阈值[18,22]。

由于 ROC 是 TPR 在所有阈值上以 FPR 为变量的函数曲线，因此 ROC 不依赖于阈值。而 AUC 是 ROC 曲线下的面积，因此 AUC 值也不依赖于阈值[18,22]。Lessmann 等人指出 AUC 对于不平衡的数据是鲁棒的[10]。AUC 的计算中自动考虑了数据中存在的不平衡。AUC 有一个统计上的解释[10]。在即时缺陷预测上下文中，AUC 可以评估预测模型对一个随机抽取有缺陷变更计算有缺陷概率比对一个随机抽取没有缺陷的代码变更计算有缺陷概率高的可能性。而在实际应用中，开发者会利用预测模型的输出结果来对工作进行调度，AUC 适合于对这种调度进行评估。

3.2.3 工作量感知(Effort-aware)指标

软件工程领域的研究者发现机器学习领域提出的性能指标不能完全满足他们对于模型实际应用时的性能评估需求。由于缺陷预测是为了辅助代码审查，在实际开发中，开发者只有有限的时间和资源用来审查代码。在有限资源下，如何使开发者检查到更多的缺陷是即时缺陷预测的关键问题。因此，研究者提出使用工作量感知(Effort-aware or Cost-effectiveness)指标对即时缺陷预测技术进行性能评估。

工作量感知指标是指当开发者根据预测模型的预测结果进行代码审查时，审查一定数量的代码(即工作量)所能检查到的缺陷数量或者比例[59]。在即时缺陷预测工作中，研究者通常将代码审查工作量设置为所有变更修改代码总行的 20%，并且提出使用在检查 20%代码的情况下来计算查准率(Precision@20%)、查全率(Recall@20%)、F1-measure(F1@20%)[19,20,25,26,27]。这些工作量感知的性能指标具体定义如下：

- Precision@20%是指检查 20%代码中找到的有缺陷变更占这 20%所包含全部代码变更的比例。
- Recall@20%是指检查 20%代码中找到的有缺陷变更占整个数据集中所有有缺陷变更的比例。
- F1@20%与 3.2.1 中 F1-measure 的定义类似，是 Precision20%和 Recall20%的调和平均，即：

$$\text{F1@20\%} = \frac{2 \times \text{Precision@20\%} \times \text{Recall@20\%}}{\text{Precision@20\%} + \text{Recall@20\%}} \tag{5}$$

这三个性能指标对有监督和无监督即时缺陷预测模型都适用。在这三个指标中，即时缺陷预测研究中最常用的性能指标是 Recall@20%[19,20,25,26,27]。Yang 等人在他们的研究中发现无监督即时缺陷预测技术在 Recall@20%比有监督即时缺陷预测技术更加出色[25]。Huang 等人和 Fu 等人复现了 Yang 等人的实验，并且使用了 Precision@20%、Recall@20%和 F1@20%对有监督和无监督即时缺陷预测技术进行评估。他们发现无监督即时缺陷预测技术在 Precision@20%和 F1@20%两个指标上效果比有监督即时缺陷预测技术效果差。这些研究表明，评估即时缺陷预测技术应当综合考虑 Precision@20%、Recall@20%和 F1@20%。

除了上述三个性能指标外，即时缺陷预测技术研究还提出了其他性能指标。Jiang 等人提出使用检查 20%代码中找到的有缺陷变更数量来对即时缺陷预测技术进行评估[20]。Huang 等人发现仅仅使用查看代码的行数并不能完整地描述开发者的工作量[26]。他们指出评估开发者工作量还需要考虑到开发者在查看 20%的代码行数时所查看代码变更的数量。开发者在查看大量的代码变更时，可能需要频繁在不同代码变更中切换，并且不同的代码变更可能会对不同文件和模块进行修改，因此审查大量变更要求开发者之间进行更多合作与交流。这种上下文切换和合作交流也会占用开发资源。同时，Huang 等人还指出如果预测模型将不存在缺陷的代码变更预测为有缺陷，开发者可能就会放弃这个预测模型[26]。因此作者提出两个新指标来对即时缺陷预测进行评估：PCI@20%和 IFA。PCI@20%是指检查 20%代码中包含的代码变更占项目中所有代码变更的比例。IFA 是指在找到第一个有缺陷变更之前，缺陷预测模型为开发者推荐错误代码变更(预测为有缺陷而实际没有缺陷)的数量。

4 即时缺陷预测技术的关键问题

虽然即时缺陷预测技术近年来获得了大量研究者的关注，取得了较大进展，但仍然存在一些亟待解决的关键问题。本章将从科学问题、技术难点和工程实践三个方面阐述目前即时缺陷预测技术中存在的问题。

4.1 科学问题

4.1.1 缺陷的可解释性

现有即时缺陷预测技术只对变更存在缺陷的可能性进行预测，而针对所预测缺陷具体是什么，如缺陷类型和位置，目前尚没有相关研究。缺陷类型刻画了缺陷产生的原因和特点[60]，缺陷位置指缺陷所在的模块、文件、函数甚至代码行，掌握缺陷类型和位置有助于辅助开发人员快速修复缺陷。尽管目前研究者已提出大量缺陷分类[60,61,62,63,64]和缺陷定位[65,66,67,68,69,70,71]技术，针对即时缺陷预测的缺陷分类和缺陷定位，目前尚没有相关研究。其原因在于现有缺陷分类和缺陷定位技术均依赖于缺陷具体信息，如缺陷报告或程序执行记录，而即时缺陷预测所预测出的缺陷由于其即时性，缺陷尚未暴露且没有相关缺陷报告或执行记录，因此，需要深入分析缺陷引入变更，研究缺陷的可解释性，并进一步研究即时缺陷定位。

4.1.2 缺陷预测结果的实用性

现有即时缺陷预测技术主要有两种类型的结果，一种是面向分类的预测结果，即每当有新的代码变更提交时，模型预测其为有缺陷或没有预测两个类别[17]，另一种是面向优先级排序的结果，即模型对新提交的软件变更集合进行排序，排在列表前面的表示缺陷可能性更高[19,25]。面向分类的预测技术常常采用分类器指标进行评估，面向排序的预测技术常常采用工作量感知的指标进行评估。然而针对软件开发或维护过程中如何有效利用即时缺陷预测结果尚没有统一的观点，因此，需要展开深入的实证研究，以调研开发者实际需要哪种类型的预测结果，如何能更有效的利用即时缺陷预测结果等问题。

4.2 技术难点

4.2.1 数据标注

如何更准确标注引入缺陷的软件变更是即时缺陷预测中的技术难点之一。近年来，研究者指出传统 SZZ 算法实现存在噪音。传统 SZZ 实现会将修改空行和注释行以及代码风格的变更、重构类变更误标记为引入缺陷变更[32,34]。针对这些噪音，研究者提出多种改进 SZZ 实现，然而这些 SZZ 实现并不能完全消除噪音[32,33,34]。例如，Neto 等人提出重构感知 SZZ 避免重构类变更被误标记为引入缺陷，但作者仍然指出该 SZZ 实现对部分重构类变更仍然无法识别[34]。同时，这些工作是基于 Java 代码对 SZZ 实现进行改进。不同编程语言在注释、代码风格和重构等方面存在差异。因此，研究者提出的改进 SZZ 实现在非 Java 项目上面临实际应用上的挑战。

4.2.2 特征提取

特征提取的准确性和多样性对模型的建立有着至关重要的影响。特征提取准确性难点在于代码变更的复杂性。主要体现在两个方面，一是代码变更具有时间戳，在计算变更特征时，需要考虑软件的动态演化，包括代码演化和开发者演化；二是代码变更提交的复杂性。软件开发和维护常采用代码版本控制系统(例如 Git)，在项目采用多个分支进行同时开发时，提取特征时对代码变更合并、关联和追溯分析进行综合考虑是一大技术难点。

特征提取多样性难点在于软件制品的多源异构特性。研究表明，软件缺陷与多种软件制品相关，包括源代码、缺陷报告、测试报告、代码审查、代码静态扫描等，提取多源特征有助于更准确的刻画缺陷，从而建立更准确的预测模型。然而上述软件制品以多源异构的形式存储在不同的软件仓库中，考虑它们之间异构性、关联性是一大技术难点。

4.2.3 模型构建

现有即时缺陷预测技术在模型构建上存在着不同的策略。研究者使用复杂的有监督机器学习模型，包括集成学习[72]和深度学习[73]，有研究者主张使用简单的无监督模型[25]。在有监督学习模型中，有研究者使

用分类模型[17]，有研究者使用回归模型[19]。不同建模技术在模型应用场景、模型输出结果上存在差异，而关于哪种建模技术是更好的即时缺陷预测模型构建技术尚没有统一的结论，因此，在模型构建技术上如何选择合适的建模技术，如何设计更先进的建模技术仍然需要进一步研究。

4.3 工程实践

尽管目前存在相关工作将即时缺陷预测引入至工业界[18,19]，然而将即时缺陷预测在工程实践中进行大规模推广仍然存在以下挑战：**(1)缺乏准确的数据环境**。在即时缺陷预测的数据标注中，识别引入缺陷的软件变更时，依赖于开发人员撰写的变更提交日志(commit log)和缺陷报告。倘若在工程实践过程中这些数据不够准确，将对模型的训练带来巨大挑战；**(2)缺乏多维的数据特征**。由于软件缺陷的复杂性，其产生机制与多源异构的软件制品相关，如源代码、缺陷报告、测试报告、代码审查报告、代码静态扫描等，在工程实践中，软件制品的多样性和准确性将直接影响模型效果。**(3)缺乏统一的评估方法**。由于目前针对即时缺陷预测技术模型评估存在着指标多样化问题，在工程实践中也将面临如何更客观的评估模型这一问题。因此，需要更多工作关注即时缺陷预测的工程实践，将工程实践与理论研究相结合，将实证研究和方法研究相结合，进一步完善即时缺陷预测技术。

5 即时缺陷预测的未来展望

针对上一章节所总结的关键问题，本章围绕数据标注、特征提取、模型构建和工程实践四个方面，展望即时缺陷预测研究的未来趋势。

5.1 加强噪音数据的处理，提出更准确的数据标注方法

实证研究表明即时缺陷预测研究中所使用的标注数据是存在噪音的。Bachmann 等人的研究表明开发者可能不会为修复缺陷的代码变更记录其对应缺陷报告 ID，从而导致部分修复缺陷的代码变更无法被识别出来[74]。这将导致这些修复缺陷的代码变更所对应的引入缺陷的代码变更也无法识别出来。Da Costa 等人在他们的工作中提出了一种框架用于评估 SZZ 算法产生数据的质量，他们使用该框架发现使用 SZZ 算法所标注的数据中存在大量噪音[33]。另外，实证研究还表明，有些缺陷在引入数年后才会被发现[75]。这意味着在对即时缺陷预测技术进行实际应用时，对于近期产生的代码变更，其中某些缺陷引入变更可能还没有被修复，因此无法被 SZZ 算法识别，这也会造成噪音数据。处理这些噪音数据，并且减小噪音数据对即时缺陷预测模型的影响，将进一步提升即时缺陷预测技术的实用性。

5.2 综合考虑多源软件制品，提取多维特征

现代大型软件项目都会使用多个系统对软件开发和维护中产生的数据进行存储和管理，例如代码版本控制系统(如 Git)、缺陷跟踪系统(如 Bugzilla)、代码审查系统(如 Gerrit)等。这些不同系统中存储着不同的软件制品，现有大部分即时缺陷预测研究都只考虑了从代码版本控制系统中提取特征。而研究表明，软件缺陷的产生是一个复杂过程，并与多种软件制品相关，包括源代码、缺陷报告、测试报告、代码审查、代码静态扫描等，综合考虑多源异构的软件制品，有助于更立体的刻画缺陷产生机制，建立更准确的缺陷预测模型。例如有研究人员已经发现结合代码审查系统，从代码审查的角度提取新的特征有助于增强现有即时缺陷预测模型的预测效果[28]。因此，综合考虑多源软件制品，提取多维特征来增强特征表征能力，将是即时缺陷预测技术的一个发展方向。

5.3 研究更先进的建模技术，取得建模技术上的突破

近年来，深度学习成为机器学习的热点研究领域，被大量应用到如图像处理、语音识别等研究中[76,77,78]。相比传统的机器学习技术，研究者发现深度学习在这些领域的应用可以取得更好的性能。Yang 等人首次在即时缺陷预测中应用深度学习技术[73]。作者使用深度学习方法对初始的变更特征进行整合从而生成更加复杂的特征，然后基于这些生成的特征构建分类器对有缺陷变更进行预测。作者发现使用深度学习技术显著

提升了即时缺陷预测模型的性能。但是，作者仅使用了深度学习中的深度信念网络[79]技术用于整合特征。而将目前机器学习领域大量研究与应用的技术，如卷积神经网络(Convolutional Neural Network，CNN)[80]应用到即时缺陷预测技术中，将有可能进一步提升即时缺陷预测技术的性能。如何有效的将这些技术应用到即时缺陷预测技术中需要未来工作进一步分析与研究。

5.4 解决实践中的工程问题，推动即时缺陷预测的广泛应用

在工程实践中，实现自动化、大规模的即时缺陷预测在数据环境准备、特征提取和模型构建等方面仍然面临许多技术难点。克服现有难点主要从以下几方面展开：(1) 积极研究大规模代码变更数据自动化标注方法，尽快构建面向即时缺陷预测的大数据环境及其演进方法；(2) 解决在工程实践中多源异构软件制品的关联问题，提出代码变更的多维特征提取方法；(3) 结合更先进的机器学习技术，提出适应于工程实践中面向大规模代码变更数据的建模方法；(4) 建立即时缺陷预测的标准评估体系，推动即时缺陷预测向更统一更准确的方向发展；(5) 为了适应云计算、移动互联网及人工智能的快速发展，有针对性的研究面向不同软件类型的即时缺陷预测技术。

6 总结

近年来，即时缺陷预测技术由于其即时性、细粒度和可追溯的优势，成为了软件缺陷预测领域的研究热点。本文围绕即时缺陷预测的数据标注、特征提取、模型构建及模型评估等方面，梳理并总结了当前研究进展。基于当前进展分析，本文总结了当前即时缺陷预测面临的关键问题及未来发展趋势。主要工作总结如下：(1) 针对数据标注，本文详细归纳了不同数据标注方法的提出背景及其优缺点；针对特征提取，本文分类并详细介绍了不同维度的变更特征；针对模型构建，本文归类了现有模型构建技术，包括有监督和无监督建模技术，其中有监督建模技术又分为同项目建模和跨项目建模；针对模型评估，本文总结了现有不同类型的验证方法和评估指标。(2) 本文从科学问题、技术难点和工程实践三个角度总结了当前即时缺陷预测面临的关键问题。(3) 本文围绕数据标注、特征提取、模型构建和工程实践四个方面展望了未来即时缺陷预测技术的发展趋势。

References:

[1] Ieee standard classification for software anomalies. In IEEE Std, pages 1–23, 2010.

[2] Michael Newman. Software errors cost us economy 59.5 billion annually, nist assesses technical needs of industry to improve software-testing. Press Release, http://www. nist. gov/public affairs/releases, (02):10, 2002.

[3] Moisey Lerner. Software maintenance crisis resolution: The new ieee standard. Software Development, 2(8):65–72, 1994.

[4] Thomas D LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In Proceedings of the 28th international conference on Software engineering, pages 492–501. ACM, 2006.

[5] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. The impact of automated parameter optimization on defect prediction models. IEEE Transactions on Software Engineering, 2018.

[6] Seyedrebvar Hosseini, Burak Turhan, and Dimuthu Gunarathna. A systematic literature review and meta-analysis on cross project defect prediction. IEEE Transactions on Software Engineering, 2017.

[7] Thilo Mende and Rainer Koschke. Revisiting the evaluation of defect prediction models. In Proceedings of the 5th International Conference on Predictor Models in Software Engineering, page 7. ACM, 2009.

[8] Xin Xia, Emad Shihab, Yasutaka Kamei, David Lo, and Xinyu Wang. Predicting crashing releases of mobile applications. In Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, page 29. ACM, 2016.

[9] Qinbao Song, Zihan Jia, Martin Shepperd, Shi Ying, and Jin Liu. A general software defect-proneness prediction framework. IEEE Transactions on Software Engineering, 37(3):356–370, 2011.

[10] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. IEEE Transactions on Software Engineering, 34(4):485496, 2008.

[11] Ahmed E Hassan. Predicting faults using the complexity of code changes. In Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on, pages 78–88. IEEE, 2009.

[12] Tim Menzies, Andrew Butcher, David Cok, Andrian Marcus, Lucas Layman, Forrest Shull, Burak Turhan, and Thomas Zimmermann. Local versus global lessons for defect prediction and effort estimation. IEEE Transactions on software engineering, 39(6):822–834, 2013.

[13] Xin Xia, David Lo, Sinno Jialin Pan, Nachiappan Nagappan, and Xinyu Wang. Hydra: Massively compositional model for cross-project defect prediction. IEEE Transactions on software Engineering, 42(10):977–998, 2016.

[14] Jaechang Nam, Wei Fu, Sunghun Kim, Tim Menzies, and Lin Tan. Heterogeneous defect prediction. IEEE Transactions on Software Engineering, 2017.

[15] Feng Zhang, Ahmed E Hassan, Shane McIntosh, and Ying Zou. The use of summation to aggregate software metrics hinders the performance of defect prediction models. IEEE Transactions on Software Engineering, 43(5):476–491, 2017.

[16] A Günes ¸Koru, Dongsong Zhang, Khaled El Emam, and Hongfang Liu. An investigation into the functional form of the size-defect relationship for software modules. IEEE Transactions on Software Engineering, 35(2):293–304, 2009.

[17] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. Classifying software changes: Clean or buggy? IEEE Transactions on Software Engineering, 34(2):181–196, 2008.

[18] Emad Shihab, Ahmed E Hassan, Bram Adams, and Zhen Ming Jiang. An industrial study on the risk of software changes. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, page 62. ACM, 2012.

[19] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. IEEE Transactions on Software Engineering, 39(6):757–773, 2013.

[20] Tian Jiang, Lin Tan, and Sunghun Kim. Personalized defect prediction. In Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, pages 279–289. IEEE Press, 2013.

[21] Shivkumar Shivaji, E James Whitehead, Ram Akella, and Sunghun Kim. Reducing features to improve code change-based bug prediction. IEEE Transactions on Software Engineering, 39(4):552–569, 2013.

[22] Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. An empirical study of just-in-time defect prediction using cross-project models. In Proceedings of the 11th Working Conference on Mining Software Repositories, pages 172–181. ACM, 2014.

[23] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. Online defect prediction for imbalanced data. In Proceedings of the 37th International Conference on Software Engineering-Volume 2, pages 99–108. IEEE Press, 2015.

[24] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. Studying just-in-time defect prediction using cross-project models. Empirical Software Engineering, 21(5):2072–2106, 2016.

[25] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 157–168. ACM, 2016.

[26] Qiao Huang, Xin Xia, and David Lo. Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on, pages 159–170. IEEE, 2017.

[27] Wei Fu and Tim Menzies. Revisiting unsupervised learning for defect predicttion. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pages 72–83. ACM, 2017.

[28] Shane McIntosh and Yasutaka Kamei. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. IEEE Transactions on Software Engineering, 2017.

[29] Audris Mockus and David M Weiss. Predicting risk of software changes. Bell Labs Technical Journal, 5(2):169–180, 2000.

[30] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In Proc. Int'l Conf. Mining Software Repositories, pages 24–28, 2005.

[31] Yasutaka Kamei and Emad Shihab. Defect prediction: Accomplishments and future challenges. In Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on, volume 5, pages 33–45. IEEE, 2016.

[32] Sunghun Kim, Thomas Zimmermann, Kai Pan, E James Jr, et al. Automatic identification of bug-introducing changes. In Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on, pages 81–90. IEEE, 2006.

[33] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. IEEE Transactions on Software Engineering, 43(7):641–657, 2017.

[34] Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. The impact of refactoring changes on the szz algorithm: An empirical study. In 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 380–390. IEEE, 2018.

[35] Thomas Zimmermann, Sunghun Kim, Andreas Zeller, and E James Whitehead Jr. Mining version archives for co-changed lines. In MSR, volume 6, pages 72–75, 2006.

[36] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. Refactoring: improving the design of existing code. Addison-Wesley Professional, 1999.

[37] Danilo Silva and Marco Tulio Valente. Refdiff: detecting refactorings in version histories. In Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on, pages 269–279. IEEE, 2017.

[38] Victor R Basili and Barry T Perricone. Software errors and complexity: an empirical investigation. Communications of the ACM, 27(1):42–52, 1984.

[39] Les Hatton. Reexamining the fault density component size connection. IEEE software, 14(2):89–97, 1997.

[40] Kyung Hee An, David A Gustafson, and Austin C Melton. A model for software maintenance. In Proc. Conf. Software Maintenance, pages 57–62, 1987.

[41] Norman F. Schneidewind and H-M Hoffmann. An experiment in software error data collection and analysis. IEEE Transactions on Software Engineering, (3):276–286, 1979.

[42] Taghi M Khoshgoftaar and Edward B Allen. Ordering fault-prone software modules. Software Quality Journal, 11(1):19–37, 2003.

[43] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. IEEE Transactions on Software engineering, 31(10):897–910, 2005.

[44] Eamonn Keogh and Abdullah Mueen. Curse of dimensionality. In Encyclopedia of Machine Learning and Data Mining, pages 314–315. Springer, 2017.

[45] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In Proceedings of the 30th international conference on Software engineering, pages 181–190. ACM, 2008.

[46] Yuanrui Fan, Xin Xia, David Lo, and Shanping Li. Early prediction of merged code changes to prioritize reviewing tasks. Empirical Software Engineering, Mar 2018.

[47] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In Proceedings of the 27th international conference on Software engineering, pages 284–292. ACM, 2005.

[48] Marco D' Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on, pages 31–41. IEEE, 2010.

[49] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction. In Proceedings of the 2013 international conference on software engineering, pages 392–401. IEEE Press, 2013.

[50] 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.

[51] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. IEEE transactions on software engineering, 33(1):2–13, 2007.

[52] Scientific Toolworks. Maintenance, understanding, metrics and documentation tools for ada, c, c++, java, and fortran. http:/www. scitools. com, 2005.

[53] Yin Zhang, Rong Jin, and Zhi-Hua Zhou. Understanding bag-of-words model: a statistical framework. International Journal of Machine Learning and Cybernetics, 1(1-4):43–52, 2010.

[54] Joel Jones. Abstract syntax tree implementation idioms. In Proceedings of the 10th conference on pattern languages of programs (plop2003), pages 1–10, 2003.

[55] Shinsuke Matsumoto, Yasutaka Kamei, Akito Monden, Ken-ichi Matsumoto, and Masahide Nakamura. An analysis of developer metrics for fault prediction. In Proceedings of the 6th International Conference on Predictive Models in Software Engineering, page 18. ACM, 2010.

[56] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pages 4–14. ACM, 2011.

[57] Patanamon Thongtanunam, Shane Mcintosh, Ahmed E. Hassan, and Hajimu Iida. Investigating code review practices in defective files: an empirical study of the qt system. In Mining Software Repositories, pages 168–179, 2015.

[58] T. Mitchell, B. Buchanan, G. Dejong, T. Dietterich, P Rosenbloom, and A. Waibel. Machine Learning. McGraw-Hill, 2003.

[59] Thilo Mende and Rainer Koschke. Effort-aware defect prediction models. In Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on, pages 107–116. IEEE, 2010.

[60] Ram Chillarege. Orthogonal defect classification. Handbook of Software Reliability Engineering, pages 359–399, 1996.

[61] Ferdian Thung, David Lo, and Lingxiao Jiang. Automatic defect categorization. In Reverse Engineering (WCRE), 2012 19th Working Conference on, pages 205–214. IEEE, 2012.

[62] Ferdian Thung, Xuan-Bach D Le, and David Lo. Active semi-supervised defect categorization. In Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, pages 60–70. IEEE Press, 2015.

[63] LiGuo Huang, Vincent Ng, Isaac Persing, Mingrui Chen, Zeheng Li, Ruili Geng, and Jeff Tian. Autoodc: Automated generation of orthogonal defect classifications. Automated Software Engineering, 22(1):3–46, 2015.

[64] Jerónimo Hernández-González, Daniel Rodriguez, InakiInza, Rachel Harrison, and Jose A Lozano. Learning to classify software defects from crowds: a novel approach. Applied Soft Computing, 62:579–591, 2018.

[65] Stacy K Lukins, Nicholas A Kraft, and Letha H Etzkorn. Bug localization using latent dirichlet allocation. Information and Software Technology, 52(9):972–990, 2010.

[66] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports. In Proceedings of the 34th International Conference on Software Engineering, pages 14–24. IEEE Press, 2012.

[67] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Transactions on Software Engineering and Methodology (TOSEM), 22(4):31, 2013.

[68] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. Where should we fix this bug? a two-phase recommendation model. IEEE transactions on software Engineering, 39(11):1597–1610, 2013.

[69] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. Bug localization with combination of deep learning and information retrieval. In Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on, pages 218–229. IEEE, 2017.

[70] Klaus Changsun Youm, June Ahn, and Eunseok Lee. Improved bug localization based on code change histories and bug reports. Information and Software Technology, 82:177–192, 2017.

[71] Thong Van-Duc Hoang, Richard J Oentaryo, Tien-Duy Bui Le, and David Lo. Network-clustered multi-modal bug localization. IEEE Transactions on Software Engineering, 2018.

[72] Xinli Yang, David Lo, Xin Xia, and Jianling Sun. Tlel: A two-layer ensemble learning approach for just-in-time defect prediction. Information and Software Technology, 87:206–220, 2017.

[73] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. Deep learning for just-in-time defect prediction. In Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on, pages 17–26. IEEE, 2015.

[74] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The missing links: bugs and bug-fix commits. In Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, pages 97–106. ACM, 2010.

[75] Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. Predicting fault incidence using software change history. IEEE Transactions on software engineering, 26(7):653–661, 2000.

[76] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. science, 313(5786):504–507, 2006.

[77] Geoffrey E Hinton. Learning multiple layers of representation. Trends in cognitive sciences, 11(10):428–434, 2007.

[78] Li Deng, Jinyu Li, Jui-Ting Huang, Kaisheng Yao, Dong Yu, Frank Seide, Michael Seltzer, Geoff Zweig, Xiaodong He, Jason Williams, et al. Recent advances in deep learning for speech research at microsoft. In Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on, pages 8604–8608. IEEE, 2013.

[79] Geoffrey E Hinton. Deep belief networks. Scholarpedia, 4(5):5947, 2009.

[80] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems, pages 1097–1105, 2012.