基于深度学习的空指针引用缺陷检测系统的设计与实现

（作 者 姓 名）

2018 年 5 月

中图分类号：TQ028.1

UDC分类号：540

基于深度学习的空指针引用缺陷检测系统的设计与实现

作 者 姓 名 （作 者 姓 名）

学 院 名 称 软件学院

指 导 教 师 （姓名、专业技术职务、学位）

答辩委员会主席

申 请 学 位 工学硕士

学 科 专 业 软件工程

学 位 授 予 单 位 北京理工大学

论 文 答 辩 日 期 2018 年 5 月

Design and implementation of null pointer reference defect detection system based on deep learning

Candidate Name:	(Author Name)
School or Department:	Software Institute
Faculty Mentor:	(Supervisor Name)
Chair, Thesis Committee:	Prof. **
Degree Applied:	Master of Science
Major:	Software engineering
Degree by:	Beijing Insititute of Technology
The Date of Defence:	June，2018

基于深度学习的空指针引用缺陷检测系统的设计与实现

北京理工大学

研究成果声明

本人郑重声明：所提交的学位论文是我本人在指导教师的指导下进行的研究工作获得的研究成果。尽我所知，文中除特别标注和致谢的地方外，学位论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京理工大学或其它教育机构的学位或证书所使用过的材料。与我一同工作的合作者对此研究工作所做的任何贡献均已在学位论文中作了明确的说明并表示了谢意。

特此申明。

作者签名：_____　　签字日期：_____

关于学位论文使用权的说明

本人完全了解北京理工大学有关保管、使用学位论文的规定，其中包括：① 学校有权保管、并向有关部门送交学位论文的原件与复印件；② 学校可以采用影印、缩印或其它复制手段复制并保存学位论文；③ 学校可允许学位论文被查阅或借阅；④ 学校可以学术交流为目的，复制赠送和交换学位论文；⑤ 学校可以公布学位论文的全部或部分内容（保密学位论文在解密后遵守此规定）。

作者签名：_____　　导师签名：_____

签字日期：_____　　签字日期：_____

摘要

空指针引用是程序中比较常见的缺陷之一，研究表明该缺陷在编译后的程序中大量存在，因此给软件的稳定性带来很大威胁。出于对检测效率和精度的平衡，现有工具在工作原理和检测范围等方面各不相同，无法全面检测该类缺陷，大量的误报也降低了此类工具的实用价值。

本文设计了一个基于 SonarQube 平台运行的插件——BIT-Detector，它集成多种静态代码分析工具对代码进行检测，然后把不同工具产生的检测报告进行交叉验证，将检测出的缺陷按照可信度优先级进行排序以提升检测报告的可信度。结果表明所有工具能够同时检测的缺陷可信度非常高，但是那些只被部分工具报告出的缺陷的真实性难以判定。为了提升 BIT-Detector 的能力，本文提出了利用深度学习技术构建模型判定不同工具检测结果可信度的方法。

为了构建训练数据集，本文通过对一些开源代码进行语法分析并在合适的地方对其语法结构进行改造来生成大量空指针引用缺陷用例。然后生成测试用例的控制流图，并对图进行了适当的改造和压缩，最后从八个维度抽取出图中每个结点的代码特征。另外，为了便于模型训练，本文将不同工具实际检测结果的准确性作为每个训练数据的标签。

利用生成的数据集，本文设计了图特征抽取模型和特征分类模型。前者可以将包含代码特征信息的控制流图转换为一定维度的向量，后者可以根据标签 (不同工具检测结果的准确性) 对这些向量进行分类。通过这两个模型，可以评估目标代码在不同工具下检测结果的置信度。

实验结果表明，相对于单一工具的检测，BIT-Detect 采用的交叉验证和缺陷可信度排序的方式大大提升了报告的实用性，某种程度上显著地降低了误报率。对于真实性难以判定的缺陷，本文设计的深度学习模型可以给出不同工具检测结果的置信度来参与决策，从而进一步提升了缺陷排序的准确性。

关键词：空指针引用缺陷；静态检测；深度学习；交叉验证；

Abstract

Null pointer dereference is one of the common defects in the program. Research shows that this defect exists in a large number of compiled programs, therefore poses a great threat to the stability of software. Due to the balance between efficiency and accuracy of detection, the existing tools can not fully detect such defects as which vary in terms of operational principle and range of detection. A large number of false positives also reduce the practical value of such tools.

This article designs a plug-in based on the SonarQube platform, BIT-Detector, which integrates multiple static code analysis tools to detect the code, then cross validates the detection reports generated by different tools, and sorts the defects by reliability to increase credibility of the detection report. The results show that the reliability of defects detected by all tools at the same time is very high, but the veracity of defects reported only by some tools is difficult to determine. In order to improve the capability of BIT-Detector, this paper proposes a method of constructing models using deep learning techniques to determine the credibility of different tools.

For building the training data set, this paper generates a large number cases of null pointer deference by parsing some open source code and adapting its grammar structure properly. Then the control flow graph of the test case is generated, and the graph is modified and compressed appropriately. Finally, the code features of each node in the graph are extracted from eight dimensions. In addition, for facilitating the training of the model, the accuracy of actual detection results of different tools is used as a label for each training data.

Using the generated data sets, this article designs a graph feature extraction model and a feature classification model. The former can convert control flow graphs containing code feature information into vectors, and the latter can classify these vectors according to the labels (accuracy of the detection results of different tools). Through these two models, the confidence level of the report from detecting target code with different tools can be evaluated.

The experimental results show that compared to a single tool detection, the BIT-Detect's use of cross-validation and defect reliability ranking method greatly enhances the practicality of the report and reduces the false positive rate to some extent. For the defects whose

authenticity is difficult to judge, the deep learning model designed in this paper can give the confidence of detection results of different tools to help make decision, thus further improving the accuracy of defect ranking.

Key Words: null pointer dereference; static detection; deep learning; cross validation;

目录

摘要 I

Abstract II

第 1 章 绪论 1

1.1 研究背景 1

1.2 国内外研究现状及发展趋势 2

1.3 本文研究内容 3

1.4 论文结构 4

第 2 章 相关工作 6

2.1 程序静态分析技术 6

2.2 静态分析技术在空指针引用缺陷检测的应用 8

2.3 代码缺陷检测工具介绍 9

2.4 深度学习技术在软件安全领域的应用 11

2.5 本章小结 12

第 3 章 总体架构 13

3.1 设计背景 13

3.2 设计思路 16

3.3 整体架构 17

3.4 本章小结 20

第 4 章 数据集的构建和预处理 21

4.1 数据集构建 21

4.1.1 数据来源 21

4.1.2 缺陷用例构造 22

4.2 控制流图提取 25
4.2.1 Soot 25
4.2.2 全局控制流图构建 27
4.3 代码特征抽取 31
4.4 数据标注 35
4.5 本章小结 36
第 5 章 深度学习模型的构建 37
5.1 图结构的数据向量化 37
5.1.1 核函数 37
5.1.2 图结构数据的核函数 39
5.1.3 希尔伯特空间 40
5.2 图结构数据的建模 40
5.2.1 马尔可夫随机场 40
5.2.2 图结构数据的建模方法 41
5.2.3 图结构数据模型的求解 41
5.3 分类模型的构建 43
5.3.1 神经网络 43
5.3.2 图结构特征抽取模型构建 45
5.3.3 判别模型的构建 47
5.4 本章小结 48
第 6 章 实验评估 49
6.1 实验环境 49
6.2 数据选择 49
6.3 训练优化实验 50
6.4 模型检测实验 51
6.5 本章小结 55

结论 56

参考文献 58

攻读学位期间发表论文与研究成果清单 61

表格

3.1 四种工具和 BIT-Detector 的测试结果对比 14

4.1 AST 中的结点信息 23
4.2 Soot 中表示项目的数据结构 25
4.3 Jimple 中 Stmt 语句类型及特征编码 32
4.4 Jimple 中调用语句的类型及特征编码 32
4.5 Jimple 中空指针传递类型及特征编码 33
4.6 控制流图压缩后的结点特征转换 35

6.1 训练集上工具检测结果 54

插图

1.1 Eclipse3.0.1 中存在的空指针引用缺陷 2

2.1 将整数空间的具体值域映射到抽象值域 8

3.1 4 种工具在 8650 个测试用例上的检测结果 14
3.2 SonarQube 平台工作流程 18
3.3 BIT-Detector 工作架构图 19

4.1 测试用例来源 21
4.2 抽象语法树示例 23
4.3 测试用例构建流程 24
4.4 Soot 工作流程 26
4.5 示例代码的 Units 关系图 27
4.6 全局控制流图构建示例 31
4.7 全局控制流图压缩示例 33

5.1 马尔可夫随机场 41
5.2 图数据隐变量建模 42
5.3 M-P 神经元模型 44
5.4 常用激活函数 44
5.5 图特征抽取模型图示 46
5.6 完整分类模型图示 48

6.1 数据集中图结点数量的分布 50
6.2 两种采样方法示例 50
6.3 迭代次数和特征维度对模型训练的影响 52
6.4 四种工具的二分类模型的分类效果 53
6.5 四种工具的二分类模型的 ROC 曲线 54
6.6 不同阈值下的缺陷判定准确率 55

第 1 章　绪论

1.1　研究背景

随着互联网的迅猛发展，各种应用软件日益增多，软件规模越来越大，但是这些软件面对的安全及其稳定性的问题也日益突出。当下人工智能技术的快速成熟，软件的智能化和自动化程度都上升到了新的高度，在越来越多的无人场景下，如无人驾驶，智慧医疗等领域，软件正逐步完全替代人类在一些重要领域的工作，这就对软件的安全性[1] 提出了更高的要求。

任何的软件设计或者编码产生的安全漏洞[2]，都可能成为潜在的安全隐患，可能会给社会带来巨大的损失，信息安全问题早就成为人们日益关注的焦点[3]。近年来，重大网络安全事件层出不穷，如 2017 年 5 月，史上规模最大的一次勒索病毒攻击事件爆发，全球近百个国家的网络遭遇 Wannacry 病毒[4] 的攻击，电脑被该病毒感染后文件会被加密锁定，支付黑客索要的赎金后才能解密恢复，受攻击对象甚至包括医院、高校等公益性机构。2013 年 6 月，前美国中情局 (CIA) 雇员斯诺登曝出一项由美国国家安全局 (NSA) 实施的棱镜计划[5]，震惊全球。据斯诺登披露的资料显示，NSA 通过植入恶意软件感染了全球超过 5 万台计算机，用于窃取敏感信息；

由于软件的复杂性随着软件的规模和数量不断增高，软件开发的难度也在增大，导致在开发过程中存在某些不确定性的错误或缺陷。另一方面软件开发人员的水平参差不齐，即使是富有经验的开发者在开发过程中也难以避免引入一些错误或缺陷。如图1.1所示，Eclipse3.0.1 中也会存在着一些明显的空指针引用缺陷[6]。公开数据显示，对于有经验的程序员编写的代码，每 1000 行就有 50-250 个错误，平均每 1000 行会有 100 个缺陷，即使是经过软件故障控制管理培训的软件工程师，平均每 1000 行代码中也会存在 50 个故障[7]。因此，整个行业在关注软件如何提升生产力的同时，也更加注重提高软件源代码编写的质量，加大了对源代码的检测力度，以期及早发现代码中潜在的安全隐患，减少甚至避免因为软件缺陷带来的损害。据统计，现在在软件开发总成本中，投入到软件测试中的资源约占到 25% 到 50%[8]，事实上，人们对于软件安全方面的关注覆盖了软件生命周期的各个阶段。

在数量繁多的软件缺陷中，空指针引用缺陷是相对比较常见的缺陷类型，广泛存在于不同的编程语言中。同时，它也是最影响软件系统可靠性和稳定性的缺陷之一。

```
if (in == null)
  try {
     in.close();
  } catch (IOException e) {}
```

图 1.1　Eclipse3.0.1 中存在的空指针引用缺陷

通过对缺陷的总结，研究人员发现常见的软件缺陷有数组越界，资源泄漏，空指针引用等。其中空指针引用缺陷出现的尤为频繁，根据 coverity 公司 2009 年针对 280 个开源项目的缺陷分析报告，空指针引用缺陷在所有种类缺陷中所占比例为 27.81%，是所占比例最高的缺陷类型[9]。

中国国家信息安全漏洞库 (CNNVD) 统计，2013 年共发现空指针引用引发的漏洞 35 个，这些漏洞存在于操作系统、服务器应用程序等软件系统中，漏洞类型有拒绝服务、代码注入、信息泄露、一区溢出、数字错误等。这些漏洞一旦被恶意攻击者利用，可能导致系统崩溃，服务器程序可能会拒绝服务，或者机密信息泄露，这都将严重的影响软件的运行以及系统的安全。根据对国内航空航天、武器装备、金融、电信等数千万行国产软件应用 DTSC 的测试报告统计，在所有故障类缺陷中，空指针引用缺陷大约会占到 30% 左右，空指针引用缺陷的密度大致是 0.3/KLOC。

总之，空指针引用缺陷的清除对于程序的稳定性和安全性都具有巨大价值，而针对空指针引用缺陷的检测技术研究也就具备了重大的意义。

1.2　国内外研究现状及发展趋势

软件缺陷检测相关的研究几乎是伴随着软件的产生而出现的，随着程序设计语言的发展，软件缺陷类型也越来越多。

以 Java 语言为例，null 关键字的广泛使用是 Java 代码中产生空指针引用缺陷的直接原因，Haidar Osman 等人开发了 NullTracker[10] 工具对 810 个开源 Java 项目进行简单的数据流分析，追踪代码中空指针检查语句的分布情况，以发现程序开发者使用 null 关键字的时机和目的。结果表明在所有的条件判断语句中，用来进行空指针检查的语句平均占比为 35%。类成员没有初始化，方法返回 null 值，以及向方法中传递 null 值是引发空指针引用缺陷的最常见因素。其中，71% 的空指针检查语句用来保证方法调用返回值的安全性。由于空指针检查语句的频繁使用，可能导致程序运行时的开销增加 2%-10%，不仅如此，空指针检查的频繁使用还会降低代码的可读性和可维

护性，而一旦缺失了这种检查，程序的稳定性便无法得到保障。

由于空指针引用缺陷在代码中广泛存在而又十分隐蔽，但是其一旦出现很大可能会导致程序崩溃，因此对程序的稳定性具有非常大的威胁。针对空指针引用缺陷的检测一直备受关注。

目前，代码缺陷检测的技术从较高层次上主要分为两大类，动态检测[11] 和静态检测[12]。

动态检测主要侧重于软件的性能、功能完善等方面。通过动态测试来进行漏洞的探测与发现，不仅仅要求测试人员对缺陷特性具有较深入的理解，测试过程中还需要大量测试用例。在目前软件规模愈加庞大，逻辑愈加复杂的情境下，这种方式必然会带来大量人力和物力的浪费。

静态检测是指利用静态分析手段来探测程序中潜在缺陷的方法。不需要运行程序，而是使用其他手段完成对程序结构的分析。静态分析技术通常会采用数据流分析技术对代码的执行情况进行抽象解释并捕捉重点关注的数据流信息，因此它具有较高的扩展性和灵活性，在较大的程序规模下也能稳定工作。

相较于动态分析技术而言，静态分析成本较低，而且能有效的对代码中的缺陷进行精确定位。因此，对源代码的静态分析和缺陷检测是一个值得深入研究的方向。

1.3 本文研究内容

静态分析技术具有较早发现缺陷、覆盖率高、低开销、自动化程序高等优点；同时静态分析技术也存在一定的局限性，不仅要在分析效率与精度中进行权衡，还需要在误报率与漏报率之间做出取舍。现有的一些静态检测工具（如 Findbugs,PMD）都采用了不同的实现方法达到这样的平衡，由于采取的分析策略的不同，形成的检测结果也往往有较大差异。

基于当下 Java 代码空指针引用缺陷检测工具的特点，本文提出了交叉验证的方法以整合不同工具的检测能力，从而提高检测结果准确率。同时利用深度学习方法，对复杂缺陷报告的判定构造决策模型。研究内容如下：

(1) 基于 SonarQube 平台开发插件，集成多种静态代码分析工具对代码进行检测，然后把不同工具产生的检测报告进行交叉验证，将检测出的缺陷按照可信度优先级进行排序以提升检测报告的准确率。

(2) 提出一种构建空指针引用缺陷测试用例的方法，并通过构造全局控制流图，抽取相应维度的代码特征的方式，用含有结点特征的全局控制流图来描述测试用例。

(3) 将含有特征信息的控制流图转换成向量，利用深度学习的方法构建神经网络模型对多工具检测报告中矛盾缺陷的真实性进行评估，进一步优化 BIT-Detector 的检测结果。

1.4 论文结构

第一章介绍了本文研究的背景，阐述了空指针引用缺陷检测的价值和意义。同时综述了国内外在空指针引用缺陷静态检测方面研究的现状和发展趋势，最后介绍了本文研究的主要内容。

第二章介绍了静态分析技术的特点和主要分析方法，并对比了这些方法的优点和不足之处。随后介绍了近年来静态分析方法在空指针引用缺陷检测的应用。本章还简要介绍了几种常用的静态代码分析工具，这些工具都可以用于 Java 代码的空指针引用缺陷检测。最后介绍了深度学习技术在软件安全领域的应用。

第三章是对本文工作进行总体的介绍，在设计背景部分，详细介绍了 BIT-Detector 设计的背景和试验效果并探讨了需要改进的地方，并引出后文深度学习模型的设计。然后介绍了 BIT-Detector 的工作环境和作业流程，并详细阐述了利用深度学习模型提升检测效果的设计思路和工作流程。

第四章介绍了模型训练所需要的数据的来源和预处理过程。首先介绍了数据的来源和利用抽象语法树生成空指针引用缺陷的方法，然后依次介绍了测试用例控制流图的生成和利用过程间调用图生成全局控制流图的方式，随后还介绍了在代码特征提取阶段为了提取合适的代码特征怎样压缩全局控制流图。最后简要说明了数据的标注方式。

第五章分别介绍了图结构特征抽取模型和特征向量分类模型的构造方式。其中对深度学习涉及的一些基本概念进行了简单介绍，阐述了图数据特征抽取模型的理论基础以及设计原理，还介绍了分类模型神经网络的连接方式。最后阐述了神经网络的训练方法以及评判标准。

第六章为实验部分，该部分首先介绍了实验的环境和数据的选取，然后对为不同工具设计的分类模型分别进行验证实验，最后使用相应数据集针对工具实际测试的结果和分类模型判定的结果进行综合实验，分析模型对于复杂检测报告缺陷真实性的鉴

别能力。

最后总结部分对论文的工作进行总结和展望，探讨本文工作的价值和需要改进的地方。

第 2 章　相关工作

为了保证软件的可靠性和稳定性，在大量研究人员长期不懈地努力下，出现了很多针对软件缺陷的检测方法。这些方法可以在软件开发周期的不同阶段介入，检测的效率和效果也大不相同，最终涌现出了一批相对成熟的代码缺陷检测方法和工具。另一方面，随着软件数量的日益庞大，以及数据挖掘技术在各个研究领域的广泛应用，利用机器学习的方式来解决软件安全问题也逐渐成为了研究热点。

2.1　程序静态分析技术

静态分析技术即是在不运行程序，不依赖程序输入的情况下对程序代码进行分析的一项技术。这种技术有助于开发人员对代码结构的理解，同时也能检测潜在的安全缺陷 (如 SQL 注入)，运行时错误 (如空指针引用缺陷) 以及部分代码逻辑错误。它一般需要配合利用自动化工具执行分析。采用的技术有数据流分析，机器学习，语义精简等。可检测死锁，空指针，资源泄露，缓存区溢出，安全漏洞，竞态条件等软件缺陷，具有快速，准确，伸缩性强等特点。能够在代码开发阶段找到并修复多种问题，从而节省大量人力成本和时间。下面对部分静态分析方法涉及的相关技术进行简要的介绍。

符号执行[13] 是静态分析中较常用到的一种技术，它可以利用抽象符号描述程序执行过程中的变量值。这种方法可以很好地模拟程序的运行过程。相对于传统方法无法确定程序真实执行下各变量值的情况，此方法在对程序进行路径敏感分析时十分有效。不过因为符号执行方法会追踪程序中所有变量的所有取值空间，所以在应用于大规模代码进行分析时，可能会导致分析的可能路径数量迅速增多，因此在应用该方法的时候，往往会采取优化路径数量的方法即选择部分可能性最高的路径进行分析，这样虽然可以避免状态爆炸的产生，但是也难免会导致分析精度的下降。

PREfix[14] 是一种针对 C 语言的静态分析工具，它采用了符号执行的方法。该工具可以对程序每个可能的执行过程进行抽象建模，静态地模拟程序的多个可能执行路径，同时利用约束求解对程序分析过程中出现的约束集合进行检查。此工具能够做到路径敏感的缺陷检查，但是由于符号执行方法的特性，为了避免状态爆炸的情况出现，它只能选取部分路径进行分析，这就导致了分析精度的不理想。

模型检查[15] 也是一种常见的静态分析技术，通常的做法是构建有限状态机或者有向图等抽象模型，再对构造出的模型进行遍历来检验待检测系统的部分性质。SLAM[16] 是一种具有代表性的基于模型检查的静态分析工具。它可以从待检测代码中抽象出一个布尔程序并加以验证。在得到的错误报告中逐个检查，找出所有误报，进而根据这些误报对抽象的布尔程序进行调优，经过不断迭代，最终可以取得很好的效果。

同样是应用于程序验证的技术，不同于模型检查，定理证明[17] 是基于语义的程序分析方法。但是由于采用了消解原理的定理证明器，而这种方法对整数域和有理数域相关的运算不是很好处理，所以应用在程序分析领域显得不是特别合适。基于这种问题，研究人员通常会选择各种判定过程来确定公式是否是定理。ESC[18] 就是一个采用了定理证明技术的半自动化工具，它在分析程序的过程中需要外界指定所涉及的类不变量，过程不变量以及循环不变量。

除了上面提到的 3 种技术，抽象解释[19] 的应用更加广泛，它是数据流分析的理论基础。1977 年 P.Cousot 和 R.Cousot 共同提出了抽象解释的理论，应用该理论分析程序时就不需要拘泥于程序最底层的具体细节上可以在更高的角度上去观察和思考。传统解释器可以知道程序中每个变量具体的值从而得到具体值域，而抽象解释不同与传统解释的地方就是可以得到一个更高阶的抽象值域。如果将一个传统解释器迁移到抽象解释器，几乎等同于构造一个函数把具体值域映射到抽象值域。如图2.1所示，我们可以将无限的整数具体值域抽象成正，负，零三个抽象值域，这个过程只需要实现一个抽象函数 α 即可完成。

抽象解释理论实际上是从代码中抽象出一些能够刻画我们想要分析的问题所需要的特征，本质上还是为了提升分析的效率而损失部分分析精度的方法。但是如果应用得当，还是可以获得很大的收益，现在几乎所有的数据流分析方法都应用了该理论。

总的来说，静态分析技术在不运行代码的情况下进行分析有效避免了程序运行环境的苛刻要求，可以针对程序的规模采取灵活的分析方法，从而具备了较早发现缺陷，较低的分析成本，较高的覆盖率和自动化程序等优点。但是由于往往需要对分析精度进行部分舍弃，导致了分析结果的漏报率和误报率都无法达到特别理想的水平。

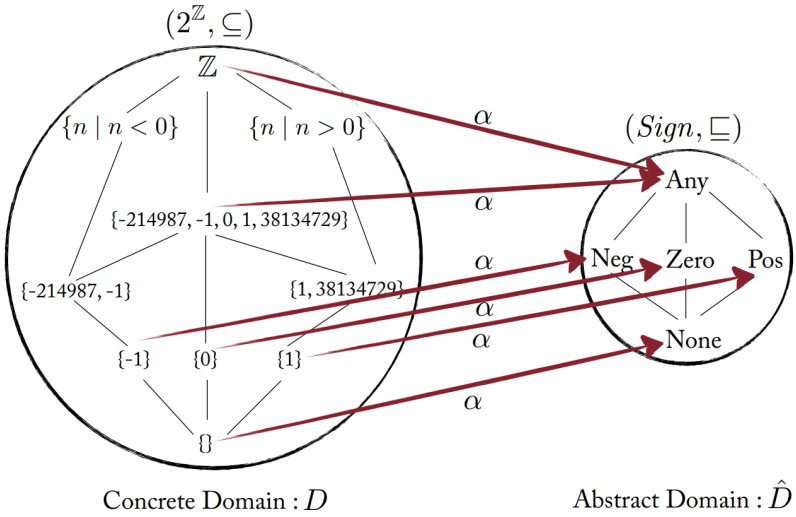


图 2.1　将整数空间的具体值域映射到抽象值域

2.2　静态分析技术在空指针引用缺陷检测的应用

针对空指针引用缺陷，研究人员已经利用静态分析技术做出了很多实践并取得了一定成果。

研究人员利用静态分析技术在 Java 空指针引用缺陷上进行了大量的工作，产生了很多检测空指针引用的工具和技术，这些技术可粗略的分为指针引用验证[20] 和空指针引用[21] 缺陷检测两大类。前者侧重于如何验证程序中的指针是否为空。后者侧重于如何尽可能多的发现程序中的空指针引用。指针引用验证技术是基于需求驱动的思想[22]，一般是首先识别出指针，再沿着控制流后向的验证指针是否为空。空指针引用缺陷检测一般是在进行数据流分析[23]、指针分析的基础上，根据一些规则基于控制流前向的检测。两者通常都需要进行数据流分析与指针分析。

Salsa[24] 是一个致力于验证 Java 代码中指针引用安全性验证的工具，通过定制的数据表示形式进行前向数据流分析，通过对传播深度和数据流传播路径数量的简单限制来获得方法的可扩展性，同时依赖预先进行的必然别名分析来提高方法间数据流分析的准确性。由于一些空指针的引用需要经过多层方法调用链才有可能触发，这种验证方式会产生很多漏报的同时，效率也不理想。数据流分析技术具有十分灵活的特点，为了提高效率，Ravichandhran Madhavan[20] 等提出了一种过近似的最弱前置条件分析方法以验证 Java 程序中指针的安全性，该方法通过需求驱动的前向数据流分析大幅提升了单个引用的分析效率，该方法试图找到程序入口处可能满足被分析程序点的引用不安全的条件，如果存在这样的条件，则可以判定该引用不安全。此方法的数据流

事实为有限的谓词集合，通过有选择地限制谓词集合的大小以及传播路径的数量，该方法可以做到低延迟的流敏感，上下文敏感的 sound 分析，利用 Wala[25] 程序分析框架，可以取得较好的验证引用安全的效果，但是过于追求针对单个引用的需求驱动分析，在对大规模代码中的引用进行批量分析时性能欠佳。

空指针检测相比于指针安全性验证更加具有实用性，而误报率和漏报率是检验工具实用性的重要指标，空指针检测工具大多不追求完美的正确率，而将较低的误报率和较高的召回率作为最重要的目标。

检测工具 Xylem[26] 从每一个指针引用出发，进行基于需求驱动的后向数据流分析，并将谓词作为数据流事实，目标是能够高效的检测出最重要的空指针引用，在进行分析时采取的是不完全可靠的分析方法，检测结果存在较多漏报。

北京邮电大学的杨睿[27] 提出一种 Java 中空指针引用故障的静态检测方法，将空指针引用问题抽象为一类故障模型，并以故障模式状态机来形式化描述此类故障模型，然后根据故障状态机的创建条件及待检测代码的语义信息确定是否创建该类型的状态机，并将创建的状态机示例置于控制流图入口，根据数据流分析的结果对故障状态进行迭代以检测空指针引用问题。

中国矿业大学的姜淑娟[28] 提出一种空指针异常自动定位方法，该方法结合程序的静态分析技术，利用程序运行时的堆栈信息指导程序切片，然后对得到的切片进行空指针分析及别名分析，得出引发空指针异常的可疑语句集合，最终给出错误定位报告。

总体来看，以上这些分析方法都有各自的优缺点，但是目前无法找到一种完美的静态检测方法可以兼顾缺陷检测的误报率和漏报率。这也正是静态代码分析的短板，不仅是将复杂的缺陷解释出来很困难，对于结果的高误报率，往往显得无能为力。

2.3 代码缺陷检测工具介绍

对于空指针引用缺陷，工业界已经产生了很多优秀的检测工具，这些工具具有不同的实现原理，对空指针引用缺陷的检测结果也不尽相同。

FindBugs[6][29] 是一个开源的针对 Java 代码的缺陷静态检测工具，通过分析 class 文件，在字节码层级进行简单的前向数据流分析，对程序中的每一个引用的是否为 null 值的不同情况，给定相应的标识从而在触发可能的空指针调用时给出不同的告警等级。对于指针引用 FindBugs 总结出了一些经验规则，对不可达路径、控制流汇合、

指针赋值语句、断言等特定情况定制了专用的检测规则，在进行过程间分析时，其主要依赖特定故障模式以及用户编码时给出的注解来推断空指针是否可能发生，所以它只能在特定场景下检测出空指针引用缺陷。

Jlint 同样是一个开源静态代码检测工具，它通过执行数据流分析和构建锁图来查找缺陷，语义矛盾和同步问题。Jlint 有两个独立的程序来执行语法和语义验证。通过使用手写扫描器和简单的自顶向下解析器，Jlint 能够检测到一些代码缺陷，例如可疑地使用操作符优先级，没有切换代码中断，对构造体错误的假设等。同时，Jlint 执行本地和全局数据流分析，计算局部变量的可能值并捕获冗余和可疑计算。通过执行全局方法调用分析，Jlint 能够检测具有可能为“null”的形参的方法的调用，并且在没有验证“null”的方法中使用该参数。Jlint 还为类依赖项构建了锁依赖关系图，并使用该图来检测在多线程程序执行期间可能导致死锁的情况。除了死锁之外，当不同的线程可以同时访问相同的变量时，Jlint 能够检测到可能的竞争条件问题。Jlint 最大的特点就是检测的效率很高，但是由于使用的数据流分析十分有限，因此误报率也较高。

Infer 是 Facebook 的开发团队在代码提交内部评审时，用来执行增量分析的一款静态分析工具，在代码提交到代码库或者部署到用户的设备之前找出缺陷。由 OCaml 语言编写的 Infer 目前能检测出空指针访问、资源泄露以及内存泄露，可对 C、Java 或 Objective-C 代码进行检测。Facebook 使用 Infer 自动验证 iOS 和安卓上的移动应用的代码，bug 报告的正确率达 80%。Infer 通过捕获编译命令，把要被编译的文件转换为可用于分析潜在错误的中间语言格式。整个过程是增量进行的，意味着通常只有那些有修改过并提交编译的文件才会被 Infer 分析。Infer 还集成了大量的构建或编译工具，包括 Gradle、Maven、Buck、Xcodebuild、clang、make 和 javac。此外，Infer 根植于两大基本理论之上，其一是霍尔逻辑，一种用于推理计算机程序正确性的形式系统，另一个是抽象解释，该理论用于测度程序语义的逼近结果，此外还涉及其它一些研究成果，例如 Separation Logic 和 Bi-abduction。

Fortify SCA 是一款应用广泛的商业工具，由知名的惠普公司出品，是一个白盒的、静态的软件源代码安全检测工具。它通过内部的五种主要分析引擎：语义、结构、控制流、数据流、配置流等对应用程序的源码进行静态分析，在分析的同时与该工具特有的软件安全漏洞规则集进行全面地查找、匹配，进而找出源代码中存在的各种缺陷和漏洞，并整理和产出缺陷报告。Fortify 应用十分广泛，在世界范围内被大量公司用作内部源代码的质量安全检测工具。

Coverity 是美国 Coverity 公司提供的可配置的用于检测软件缺陷和安全隐患的静态源代码分析解决方案，该工具基于布尔可满足验证技术应用于源代码分析引擎，分析引擎利用其专利的软件 DNA 图谱技术和 meta-compilation 技术，综合分析源代码、编译构建系统和操作系统等可能使软件产生的缺陷。Coverity 是第一个能够快速、准确分析当今的大规模、高复杂度代码的工具，它解决了影响源代码分析有效性的很多关键问题，如编译兼容性，构建集成，高误报率，有效的错误根源分析等。

2.4 深度学习技术在软件安全领域的应用

深度学习是人工神经网络中一种多层级学习框架，试图通过构建深层网络模拟人脑感知抽象概念的能力。近年来，深度学习凭借着强大的特征学习能力，问题表达能力，数据容纳能力，掀起了又一次机器学习的浪潮，并在计算机视觉，语音处理，自然语言处理等众多领域取得了巨大进展，受到从学术界到工业界的广泛关注。现在，深度学习技术也开始渗透进软件工程的多个领域。

由于代码作为输入数据的特殊性以及复杂性，深度学习在软件安全方面常见的应用方法有两种。一种是以自然语言处理的思想来挖掘代码里的潜在信息；一种是将代码抽象为控制流图，以控制流图作为输入，压缩控制流图为向量之后进行分类回归运算。

Martin White[30] 等人提出了一种自然语言处理算法检测代码克隆的方法，该方法使用了两个 RNN(递归神经网络) 模型。先运用语法分析对代码进行预处理，然后用第一个 RNN 网络得到中间向量，再对代码进行词法分析得到代码的抽象语法树，将中间变量和语法树作为第二个 RNN 网络的输入，得到最终的检测结果。这种方法考虑到了代码文本上的相似性，忽视了代码结构上的相似性。

Hanjun Dai[31] 提出了一种基于神经网络的图特征抽取的算法，能够根据不同的分类标准将代码控制流图压缩为多维的向量。Xiaojun Xu[32] 改进了该网络，通过将两段代码的控制流图成对的进行训练，从而检测两个二进制代码的相似度。通过该网络可以预测待测代码与已知缺陷代码的相似度，从而判断该代码是否为缺陷代码，为神经网络在检测代码缺陷方面的应用提供了新的思路。

2.5 本章小结

本章首先介绍了静态分析涉及的相关技术背景，然后以空指针引用的检测为例，介绍了国内外研究人员利用静态分析技术在空指针引用缺陷检测方面的进展，随后对一些业界成熟的静态代码缺陷检测工具，如 Findbugs，Jlint，Infer，Fortify 等进行了简要的介绍，最后对当下热门的深度学习技术在软件安全领域的应用进行了讨论。

第 3 章　总体架构

本文的目标是为了改善当前可用的静态代码工具误报率过高，产生的检测报告实用性较差的困境。因此基于 FindBugs，Jlint，Infer，Fortify 等工具设计了一个基于 SonarQube 平台运行的插件，该插件使用交叉验证的思想对四种工具产出的报告进行可信度排序，并利用深度学习技术训练的模型来进一步提高排序的准确性。

3.1　设计背景

随着软件规模的增大，缺陷检测的难度和所需要的代价都越来越大。就 Java 语言的空指针引用异常的检测来说，目前业界存在着很多工具。如 Findbugs，Jlint，Infer，Fortify 等。他们在检测缺陷时使用了模式匹配，数据流分析，类型系统，模型检查等技术。由于不同的技术出于对检测精度和效率的权衡，他们所产出的检测报告往往各不相同，并且几乎都包含了大量的误报和漏报。开发人员在面对这样复杂的报告时，很难判断某条报告的准确性。NickRutar[33] 等人针对五种 Java 语言的缺陷检测工具做了比较，发现没有任何一个单一的工具是完美的。此外，不同工具所产出的报告之间也有不小的差异。

基于这种情况，可以设想将多种工具的报告汇总到一起进行交叉验证。如果多个工具同时给出了同一个位置出现同一种缺陷的报告，则有理由相信这个缺陷是真实可信的。因此，本文基于 SonarQube 平台开发了插件 BIT-Detector。这个插件集成了 Findbugs，Jlint，Infer 和 Fortify 的检测能力。针对同一份待测代码，首先使用四种工具分别检测并给出报告，然后过滤出报告中的空指针引用缺陷，最后将四份关于空指针引用缺陷报告的格式统一化并进行比对，将不同工具同时检出的空指针引用缺陷作为 BIT-Detector 的输出。

由于难以找到合适的空指针引用缺陷数据集，为了对这些工具进行合理的评测，本文采用一种特别的方式构建了一批可信的测试用例，这些用例的构造方法会在后面的章节详细说明。利用构建出来的 8650 个测试用例，我们针对上文提到的四种工具以及 BIT-Detector 进行了测试。图3.1反映了各个工具检出的空指针引用缺陷的重叠情况，表3.1给出了不同工具检测的精度信息，同时还给出了 BIT-Detector 的数据。通过对比不难发现, 各个工具的检测结果确实有较大差异。即使我们使用检测准确度最高

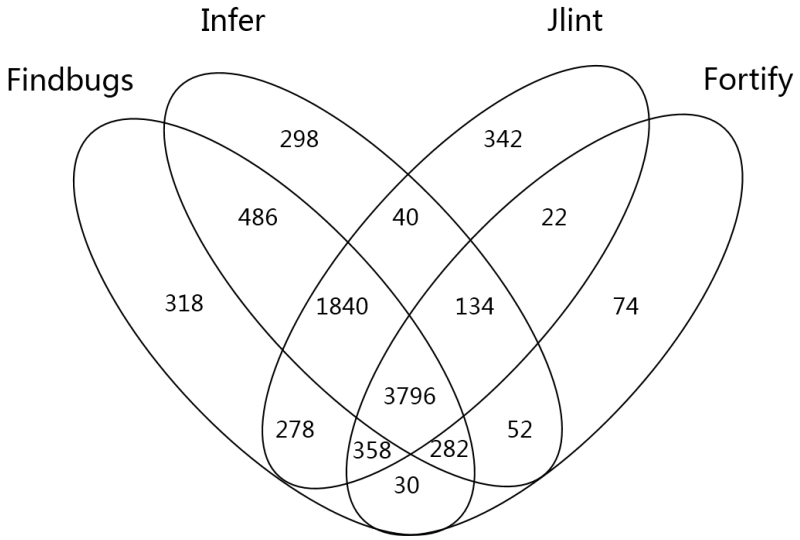


图 3.1　4 种工具在 8650 个测试用例上的检测结果

表 3.1　四种工具和 BIT-Detector 的测试结果对比

测试工具	正报	误报	准确率	召回率
Findbugs	5402	3169	63.0%	62.5%
Jlint	4933	9137	35.1%	57.0%
Infer	5051	3578	58.5%	58.4%
Fortify	3766	2211	63.0%	43.5%
BIT-Detector	3404	576	85.5%	39.4%

的 Findbugs，也会面临超过三分之一的误报。这些误报掺杂在检测报告中会给开发人员的缺陷修复带来很多困扰，很多时间和精力都会被浪费在验证缺陷的真实性上。而保证报告的准确性应该是对检测工具的基本要求，即使不讨论准确率，过多的漏报也会让人沮丧。显然，目前被广泛使用的各种检测工具还有很大的提升空间。

另外，从检测结果上可以发现，以所有工具共同检出的缺陷作为输出结果的 BIT-Detector 在准确率方面有着突出表现。相对于表现最好的 Findbugs 和 Fortify，BIT-Detector 有着 22% 的准确率提升, 而相对于准确率较低的 Jlint 而言，BIT-Detector 的准确率提升则是成倍的增加。即使召回率的表现不尽人意，但是如果不对交叉验证的结果进行过滤，而是将四种工具的检测结果按照优先级排序，召回率在某种程度上反而是提升的。如果按照这种方式处理检测报告，此时 BIT-Detector 的检测结果将自然地排在最前面，但是对其他结果的排序就成了一个棘手的问题。

假如我们有四种工具，分别记为 T_a，T_b，T_c，T_d。如果工具 T 在被测代码的 L 位置成功检出了某个缺陷 D，则记为 $E(T, L, D) = 1$，反之则记为 $E(T, L, D) = -1$。给

定位置 L_i，存在如下这种情况

$$\begin{cases} E(T_a, L_i, D_i) = 1; \\ E(T_b, L_i, D_i) = 1; \\ E(T_c, L_i, D_i) = -1; \\ E(T_d, L_i, D_i) = -1 \end{cases}$$

在这种情况下，根据交叉验证后的报告，开发人员很难判断位置 L_i 处是否存在缺陷 D_i。由上文已知，不同工具对同一份代码的检测结果不同，且它们的检测能力往往不能互相覆盖。但是，对于缺陷 D_i, 如果已知工具 T_a，T_b，T_c，T_d 对该缺陷的检测能力，那么对于检测结果的可信度就可以依据不同工具对于该缺陷的检测能力来确定。假设一个工具对于某种缺陷的检测能力可以量化为 [0-1] 区间的某个值，这里称作置信度。在这个例子中，如果工具 T_a 和工具 T_b 对该缺陷的检测结果置信度都为 0.8，而工具 T_c 和 T_d 对该缺陷的检测结果置信度都为 0.3，显然该缺陷为真实缺陷的可能性就大大增加。如果能量化得到每个工具对每个缺陷检测结果的置信度，就能量化出该缺陷为真实缺陷的可能性，从而可以对缺陷进行排序。

对于不同的缺陷类型，如空指针引用缺陷，资源泄露缺陷，跨站脚本引用缺陷等，对不同工具进行能力评估是比较容易的，只需要计算不同工具在特定类型缺陷上的准确率就能大致评估在该类缺陷上不同工具的检测能力。但是如果只限定在单一的缺陷类型上，例如缺陷 D_1，D_2 都是空指针引用类型的缺陷，确定不同工具对缺陷 D_1，D_2 的检测能力就是比较棘手的事情了。可能需要找出缺陷 D_1 和 D_2 在代码结构和语义上的不同之处，对其进行分类，还需要明白不同工具在检测策略上对这两种缺陷代码处理的不同之处才能准确地评估它们对于这两种缺陷的检测能力。

显然，利用人工去分类所有空指针引用缺陷是相当困难的事情，分类的种类是否全面，粒度是否精确都会对评估不同工具检测能力的结果造成很大影响。面对这种情况，深度学习具备相当不错的解决方案，如前文相关工作中提到的，研究人员已经利用深度学习方法在代码的分类上取得了相当不错的效果。受此启发，本文将利用深度学习的方法评估不同工具对空指针引用缺陷的检测能力。

3.2 设计思路

从上一节设计背景中的讨论可知，不同工具对空指针引用缺陷的检测能力是不同的，而这种能力也并不能很容易地进行评估，这就造成 BIT-Detector 不能对多个工具的检测报告进行很完美的交叉验证，即在检测报告中对于同一位置的缺陷，不同工具可能会给出不同的判定结果。本文提出使用深度学习的方法，在大量缺陷代码数据集的基础上，利用深度神经网络根据代码和工具检测的结果训练出判定不同工具对特定代码检测能力的模型。

深度学习的方法需要大量的训练数据才能得到表现良好的模型，所以首先应该解决开源空指针引用缺陷用例缺乏的问题。然后为了方便后续的模型训练，需要将代码的结构特征转化为数学空间的向量特征，在这个过程中还要解决代码的控制流提取以及代码特征抽取问题。最后需要给这些数据打上标签，标签应该体现出不同工具对该缺陷的检测能力。最后将标注过的数据作为模型的输入，完成整个训练过程。详细思路如下：

(1) 数据集构建

数据集的构建是模型训练所必须的工作基础，在开源项目中很难找到可用的空指针引用缺陷用例。为了达到良好的训练效果，用例应该具备完整的语义，正确的逻辑，多样的代码结构，至少一条可达路径上必然会出现空指针引用缺陷等特点。考虑到后期的处理和训练，作为用例的程序不应过大，恰好包含满足空指针引用缺陷产生的上下文最佳。加上这些要求，在开源环境下直接获得可用的训练数据集就更加困难。

考虑到 LeetCode，ACM 等具备竞赛性质的项目下可以获得大量代码完整，逻辑正确同时工程精简的代码集，数据集的构建可以通过改造这些代码来完成。为了增加代码的多样性，还可以加入 OWASP[34]，NIST[35] 等跟软件安全相关的项目下的数据集，另外也可以通过人为构造部分数据集作为补充。

(2) 控制流图生成

代码的结构化特征可以通过控制流图很好的表达出来，不同程序之间的结构化差异很大程度上可以通过控制流图的结构反映出来，所以生成正确的程序控制流图是后续模型训练的重要保证。

在具备相当数量的缺陷用例数据集后，需要将这些用例的控制流图提取出来。这个过程可以利用现有的工具来完成，例如 SOOT，LLVM 等工具，它们可以很方便地构造出 Java 代码的控制流图和方法间调用图。利用这两方面的信息，基本可以提取

到代码所有的结构化特征。从本文关注的角度来看，只需要提取出空指针缺陷发生的上下文信息即可，即提取从变量赋值为 null 到变量被解引用产生异常为止的控制流子图。

(3) 代码特征抽取

控制流图只能反映出程序的结构化信息，对于空指针引用缺陷的检测来说，工具需要更多地理解代码的语义信息，因为一个赋值语句的差别就可以很容易地决定是否能够产生空指针引用缺陷。如果要体现程序的语义信息，需要尽可能多地抽取出每一句代码的静态特征，如是否将 null 赋值给了变量，是否针对 null 值进行了检查，是否调用了其他方法等。这些特征将附着在控制流图的每一个结点上，为后续的模型训练提供更多的代码语义特征信息。这些工作在生成的控制流图的基础上完成。

(4) 数据标注

对于一个给定的程序片段，模型理想的输出结果为不同工具检测该段代码的能力，即这些工具检测结果的置信度。那么本方案中数据的标签可以为不同工具对该程序的实际检测能力，只要获得该程序包含缺陷的准确信息，数据标注工作就可以利用不同工具批量检测完成。

(5) 模型训练

模型训练大致可以分为两部分，第一部分模型负责将上述步骤中产生的包含特征向量信息的代码控制流图向量化。第二部分将这些向量化后的图向量和相应的标签一起进行分类训练，对于不同的工具，训练出不同的模型，这些模型都可以针对给定的代码输出该工具的置信度。

(6) 结果验证

最后，可以利用不同工具对应模型输出的置信度，结合这些工具实际的检测结果得到缺陷为真的可能性，从而优化检测报告的排序。

3.3 整体架构

本节简单介绍一下整个检测系统的运行模式。如图3.2所示，SonarQube 平台可以非常方便地集成开发人员使用的 IDE，代码版本管理工具以及持续集成框架。整个系统的工作流程大致如下：

(1) 用户在 IDE 中完成代码编辑之后推送代码至 SCM（Software Configuration Mangement）仓库，选用的工具可以是 SVN，Git 等。

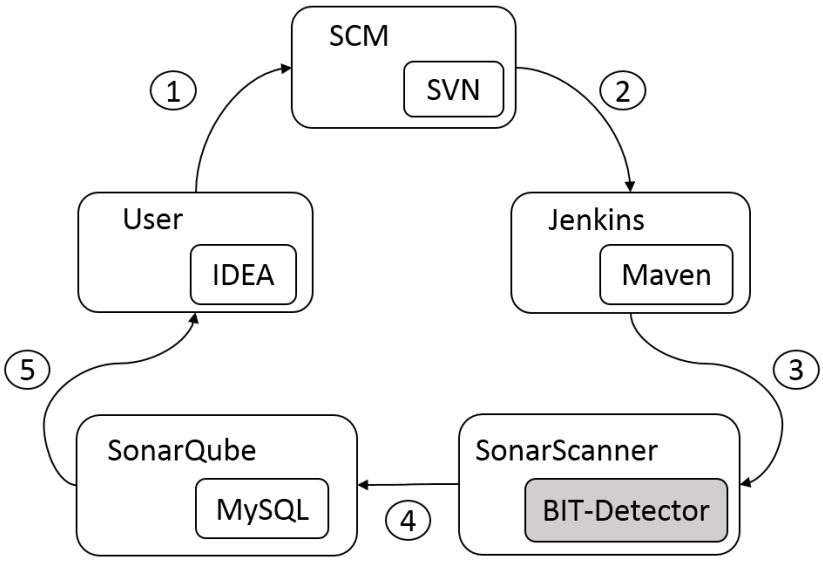


图 3.2　SonarQube 平台工作流程

(2)Jenkins 通过配置可以与核心仓库中的代码进行同步，并能感知仓库代码是否生了变动。

(3)Jenkins 在感知到核心仓库的代码发生改变之后，自动触发代码构建动作，构建工具可以选用 Maven，Gradle 等。代码成功构建之后，Jenkins 会触发 SonarScanner 扫描代码的操作。

(4)SonarScanner 首先加载配置文件记录的代码位置及目录结构信息，然后由插件负责对代码进行全面检测，并生成检测报告。最后检测报告将按照特定格式传递给 SonarQube 服务器。

(5)SonarQube 服务器会处理 SonarScanner 传递的代码检测报告，并将报告数据存储在 MySQL 等数据库中，一个工程同时可以拥有多份不同批次的检测报告，最后这些报告将会以合适的方式可视化给用户查阅。

整个环境的搭建工作并不复杂，本文重点介绍 SonarScanner 中插件 BIT-Detector 的部分设计开发工作。BIT-Detector 在 SonarScanner 扫描代码的过程中承担着检查代码的工作。它的主要工作流程如图3.3所示。

首先，用户需要针对特定的工程配置好 sonar-scanner.properties 文件。文件中记录了工程的名称，SonarQube 项目的代号，版本等基本信息，还有工程的源码，测试代码及编译后的字节码文件所在目录和依赖库等信息的位置。这些信息将提供给插件中的代码检测工具使用。在获得必要的信息之后，调用不同的工具对代码进行检测，然后根据不同工具的工作特点收集相应的检测报告。不同的工具所产生的报告格式是不

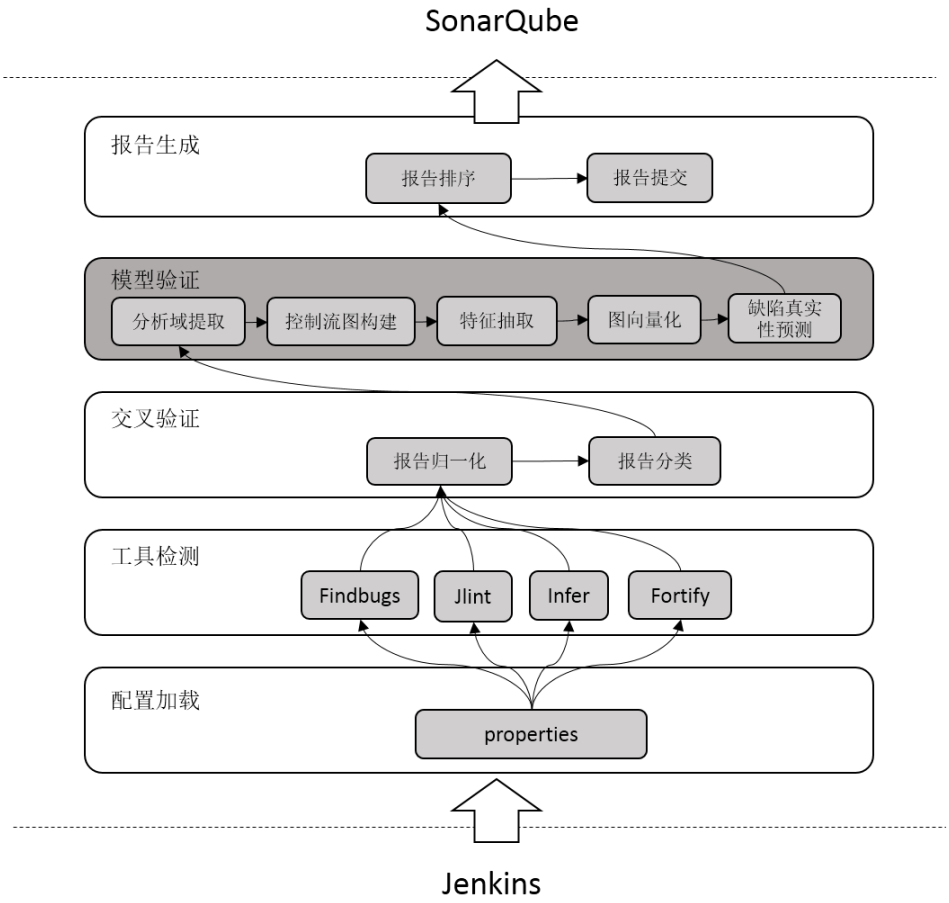


图 3.3　BIT-Detector 工作架构图

一样的，在交叉验证阶段，需要将这些报告的格式统一处理，同时还需要过滤掉不需要的缺陷类型。例如本文重点关注空指针引用缺陷，只需要留下空指针引用缺陷类型即可。然后可以根据汇总的缺陷数据对检测报告进行粗略的分类，选取所有工具都能检测出的缺陷作为最高优先级的缺陷，其他缺陷需要经过模型进行进一步的验证。在模型验证阶段，首先需要确定用例的分析域范围，即空指针产生的上下文，这些信息可以通过代码检测工具得到。然后依次提取控制流图，抽取代码特征，将控制流图向量化，最终通过模型获得不同工具检测的置信度。利用工具的置信度可以得到实际检测结果为真的可能性，据此对缺陷进一步排序。最后将报告提交给 SonarQube 进行可视化输出。

利用模型评价工具的检测能力并预测缺陷真实性是本文重点研究部分，后续的章节将会重点介绍模型训练相关的工作，其他关于插件开发的工作不再过多阐述。

3.4 本章小结

本章介绍了 BIT-Detector 的设计背景。随后详细讨论了 BIT-Detector 的设计思路，特别交代了在工具中引入深度学习方法的原因。之后介绍了基于 SonarQube 平台的代码检测流程，重点介绍了 BIT-Detector 的架构设计和工作过程，最后阐明全文研究的重点聚焦在深度学习模型的训练上面。

第 4 章　数据集的构建和预处理

上一章节提到模型的构建需要大量的测试用例，由于合适开源用例的稀缺，合理利用现有无缺陷代码来生成空指针引用缺陷是一种可行的方式。为了便于后续的处理工作，测试用例的选择应该从多维度慎重考虑。然后，为了模型训练的顺利进行，还需要对这些数据进行预处理，构建出正确的控制流图，提取出合适的代码特征。这些工作都是神经网络模型训练所依赖的重要基础。

4.1　数据集构建

4.1.1　数据来源

如果采取从正常代码中构造空指针引用缺陷的方式，首先面对的问题就是选择构建用例的合适的代码资料。为了便于后期处理，用例应该包含程序入口，具备语义完整，结构多样化，代码规范简洁等特点，只要程序包含常见的语法结构和调用关系，程序规模不应过大，恰好包含空指针引用缺陷产生的上下文最佳。依据这些条件，本文选择了部分开源代码数据集作为构建空指针引用缺陷的原始资料，如图4.1所示。

其中，OWASP 和 NIST 分别为代码缺陷检测相关领域的项目。从这些项目下可以获得部分标准的空指针引用缺陷用例，同时也可以得到很多具备其他缺陷的测试用例。由于这些用例的代码编写较为规范并经过了合理分类，还往往包含说明文档等辅助理解代码的信息，大多都可以用来生成空指针引用缺陷。除此之外，LeetCode 和 ACM 作为编程竞赛性质的项目下也包含大量可以利用的代码，这些代码根据题目的

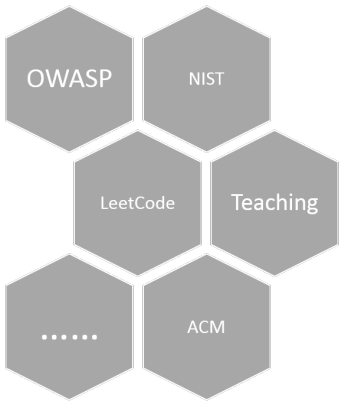


图 4.1　测试用例来源

难易级别具备着不同的复杂程度，包含了多样的代码结构，同时，这些代码的规模往往不大，是作为空指针引用缺陷用例构造的良好资源。另外，一些供教学使用的代码也是十分合适的空指针缺陷构造来源。作为补充，本文还添加了部分人工编写的测试用例。

4.1.2 缺陷用例构造

在取得构造缺陷用例的原始代码后，需要对这些代码进行检查，确保代码有正确的程序入口，并且可以正确执行。对于 OWASP 和 NIST 项目下的代码，很多用例缺乏 Main 方法的入口，这会导致后续控制流图提取的困难，这时需要在源代码层级加入合适 Main 方法，调用合适的方法驱动程序的执行。此外，对于 ACM 和 LeetCode 项目下的代码资源，虽然所有用例都包含正确的程序入口，但是往往需要正确的标准输入流数据程序才能正确执行。获得这些用例的输入数据并不困难，只需要按照相应题目下的输入输出样例给予输入数据即可驱动程序执行，这些数据的获取可以通过爬虫程序取得，自动化输入数据则可以通过重定向程序的输入流来完成。

空指针引用缺陷的产生必然需要一个产生 null 值的缺陷源。在程序的某个位置，变量被赋值为 null，随后沿着控制流图向前传播，在遇到对该变量的解引用时便会触发空指针引用异常。这个变量便是缺陷源，只要在程序中的合适位置构造缺陷源，则有一定的可能会在程序的下文中触发空指针引用缺陷。当原始代码的可用性得到确认后，需要对程序进行语法分析，寻找合适的空指针产生点并构造缺陷源。

对 Java 文件进行语法分析可以使用 Eclipse JDT 下的 AST 来完成，该工具可以在 Eclipse 环境下获得。利用它能够对 Java 文件进行解析，生成相应的抽象语法树，并且能够任意修改 Java 代码的语法结构。在 AST 中，Java 代码的每一个语法结构都有对应的 AST 结点表示，这些结点具有完整的层次关系，可以表示整个程序对象到具体方法的某个具体变量。如图4.2所示，一个 for 循环的代码片段按照 Eclipse AST 的标准解析出抽象语法树, 表4.1表示部分结点在 AST 树中对应的名称。

在生成用例的抽象语法树后，只要找到合适的点位，就可以通过修改相应的操作数为 null 来构造空指针引用缺陷源。通常这些点位都和赋值表达式有关，但是在过程间调用的上下文中，方法的参数和返回值都可以是合适的构造点位。可以利用的修改位置如下：

(1) 类的属性成员。

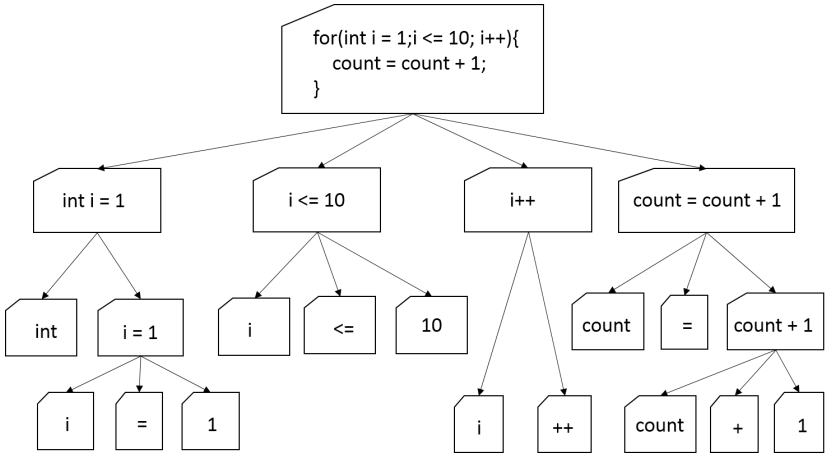


图 4.2　抽象语法树示例

表 4.1　AST 中的结点信息

子节点	子节点名	依附于父结点的角色
int i = 1	VariableDeclarationExpression	INITIALIZERS
i <= 10	InfixExpression	EXPRESSION
i++	PostExpression	UPDATERS
count = count + 1	Block	BODY

(2) 方法内的局部变量。

(3) 方法的参数。

(4) 方法的返回值。

其中 (1) 中的属性成员包含被初始化的非 null 的普通属性和静态属性。(1) 和 (2) 需要找到相关的赋值表达式，通过修改右操作数为 null 来生成空指针引用缺陷源。(3) 需要判断被调用方法的参数列表中属性的类型，将引用类型的实参修改为 null 即可。(4) 需要判断该方法的返回值类型，只有返回值为引用类型才可以修改。

图4.3为空指针引用缺陷用例构建的流程图，在抽象语法树的基础上进行修改获得缺陷源后，需要对程序进行编译并执行才能确定能否真正构建出空指针引用缺陷用例。如果没有通过编译或者运行后没有发生空指针引用缺陷，则用例构造失败，需要重新寻找新的构建点位。重复此步骤直到成功产生空指针引用缺陷。最后，成功构建的缺陷用例需要在代码中添加代码信息的注解表明该缺陷的缺陷源和发生空指针解引用的位置。运用注解的方式是为了后续代码信息抽取的工作顺利进行，因为 Java 程序可以很方便地抽取代码的注解信息，而这些自定义注解不会对代码的实际语义产生影响。记录空指针解引用的位置是为了验证工具检测的结果，加上缺陷源的位置可以

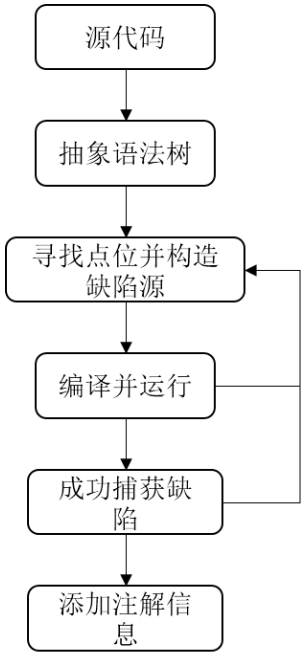


图 4.3　测试用例构建流程

很方便的确定该空指针引用缺陷发生的上下文范围，确定分析域。这些注解信息需要放置在测试用例的 Main 方法所在类中，方便代码信息抽取时统一处理。如果缺陷的产生位置不在 Main 方法所在类的方法中，就需要在注解信息中表明该缺陷产生位置所在的类。

下面的代码片段就展示了一个成功生成的测试用例，在代码的第 8 行将 $sb = newStringBuilder(\text{“”})$ 修改为 $sb = null$，随后在代码的第 11 行即对 sb 进行解引用操作，触发空指针引用缺陷。在第一行的注解标注了该缺陷涉及的上下文代码行号及变量名，这表示了分析域的范围。这个例子非常简单，实际上产生的代码在复杂度上各不相同，选用的原始代码往往都具备跨方法和跨文件的调用关系。

```
1  @Context(start = 8, end = 11, var = "sb")
2  public static void main(String[] args) {
3      Scanner in = new Scanner(System.in);
4      int k = in.nextInt();
5      if(k > 36){
6          System.out.println("-1");
7      }else{
8          StringBuilder sb = null; // source
9          int mul = k/2;
10         while(mul-- > 0){
```

```
11                  sb.append("8"); // npe
12              }
13              if(k%2 == 1){
14                  sb.append("4");
15              }
16              System.out.println(sb.toString());
17          }
18      }
```

4.2 控制流图提取

控制流图（Control Flow Graph，CFG）是一个程序或者过程的抽象表现，代表了程序执行过程中所有可能经历的路径信息，能准确刻画程序的结构信息。空指针引用缺陷测试用例生成完毕后，需要生成程序的控制流图。

4.2.1 Soot

Soot[36] 是一种 Java 字节码优化框架，凭借着对 Java 语言强大的分析能力，已经被广泛地应用于很多针对 Java 语言的分析优化项目。Soot 框架最大的特点是提供了四种不同的代码中间表示形式：Jimple，Baf，Grimp 和 Shimple，它们考虑到不同的分析场景，对代码进行了不同程度的抽象表示。同时，Soot 还构建了自定义的数据结构来表示待分析的项目，这些自定义类型与 Java 代码中的层次结构一一对应。如表4.2所示。这种表示方法使得代码的分析过程变得更加简单和灵活，易于理解使用。Soot 的工作过程如图4.4所示。

表 4.2　Soot 中表示项目的数据结构

类名	描述
Scene	表示整个分析环境
SootClass	表示一个 class
SootMethod	表示一个 Method
SootField	表示一个类成员属性
Body	表示一个方法体

Jimple 是 Soot 框架中最主要的代码中间表示形式，采用了典型的基于三地址的语句表达形式，它可以由 Java 源代码或者 Java 字节码转换得到。其中，通过字节码

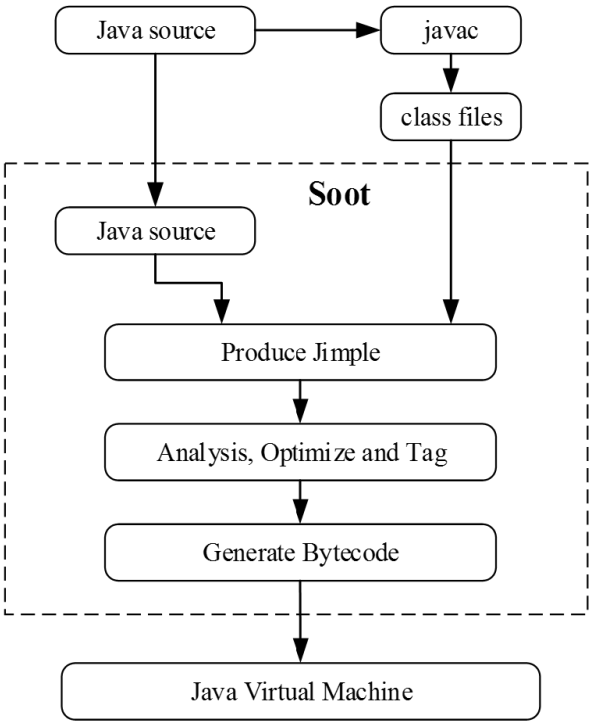


图 4.4　Soot 工作流程

的转换通过为隐式的堆栈变量引入新的局部变量以及清除 jsr 指令来完成。在代码转换的过程中，Jimple 中的局部变量会被加上被推断出来的类型信息，同时部分没有用到的局部变量也会在优化阶段被清除掉从而不会出现在 Jimple 代码中。最重要的是代码中所有的表达式都会经过线性化的处理，最终 Jimple 将由若干三地址表达式组成。这些表达式至多包含三个变量或者常量信息，这种规则对于执行代码优化是非常方便的。另一方面，Jimple 表示形式所包含的语句类型只有 15 条，相比于 Java 字节码中多达 200 多条的指令类型，Jimple 的表示形式是足够简洁的，这些精简之后的指令类型也给本文后续的代码特征提取提供了极大的便利。

除了简洁准确的表示形式，利用 Soot 还可以进行复杂的别名分析和数据流分析。另外，每个待分析方法的方法体，即表4.2中的 Body，都包含了一个 Units 链。Unit 是 Soot 中的一个接口，具体实现则是一个三地址表达式，这些表达式实际上会对应到 Jimple 中的某个具体语句类型，即 Stmt。这些 Unit 已经包含了互相之间的结构关系，根据这些结构关系可以很方便地生成方法内的控制流图。同时，对于包含调用方法语句类型的 Unit，通过 Soot 提供的 API 还可以很方便的得知该调用的对象方法，这对后面过程间调用图的构建也非常重要。

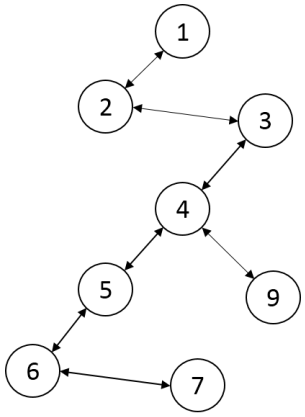


图 4.5　示例代码的 Units 关系图

4.2.2　全局控制流图构建

上一节介绍了 Soot，利用它可以构建出方法内的控制流图和过程间的调用图，进而得到全局控制流图。

方法内的控制流图可以从以 Jimple 表示的方法体中包含的 Units 链得到。通过遍历 Units 链中的 Unit 元素，可以得到它们的前驱和后继结点，这些信息实际上就表示了该方法的控制流图，不过它的结点是一个 Jimple 表示的 Stmt，在 Jimple 中表示了一种语句类型。如下面的 Jimple 代码片段所示，不同的语句对应着 Jimple 中不同的 Stmt 类型，这些类型有 15 种之多，是 Jimple 能够表示的最基本的语法单位。

```
1  public int foo(java.lang.String){// locals
2      r0 := @this; // IdentityStmt
3      r1 := @parameter0;
4      if r1 != null goto label0; // IfStmt
5      $i0 = r1.length(); // AssignStmt
6      r1.toUpperCase(); // InvokeStmt
7      return $i0; // ReturnStmt
8  label0: // createdbyPrinter
9      return2;
10 }
```

从该代码块中可以抽取出 foo 方法包含的 Units 链，进而得到其结构信息，如图4.5所示，结点的序号为代码块中语句对应的行号。从图中任意一个结点都可以找到其相邻的前驱和后继结点，这种关系不是静态代码顺序的关系，其考虑了代码执行逻辑的语序。

过程间调用图的构建需要依赖方法体内的方法调用语句，如上文代码块中的第 6 行语句，该 Stmt 的类型为 InvokeStmt。实际上 InvokeStmt 是一个接口，在 Jimple 包含的 Stmt 中，实现 InvokeStmt 接口的类有五种，它们表示五种调用方法的类型：

(1)invokestatic：调用静态方法。

(2)invokespecial：调用实例的私有方法、父类方法或构造器方法。

(3)invokevirtual：调用实例中的虚方法。

(4)invokeinterface：调用接口方法。

(5)invokedynamic：调用动态方法。

其中，前四种调用方式可以由静态分析得到，而 invokedynamic 调用方式所调用的目标必须在运行时才能动态确定，所以本文只需关注前面四种调用方式，通过对调用目标的解析建立方法之间的调用关系，从而构建出完整的过程间调用图。

这里定义过程间调用图为 G，G 包含两个数据集合 $Caller(U, S_m)$ 和 $Callee(M, S_u)$，前者表示 Unit 结点与其调用的 Method 集合的映射关系，后者表示 Method 与调用它的 Unit 结点集合的映射关系。由于 Unit 和 Method 在 Soot 分析环境中具有唯一性，并且 Unit 和 Method 还包含了其结构的上下文关系信息，即利用 Unit 可以得到其所属的 Method 信息，甚至 Class 信息，所以不需要构建 Method 与 Method 之间的调用关系。利用这两个数据集合可以方便地找出某个 Unit 调用的 Method 集合 S_m 以及调用 Method 的 Unit 集合 S_u。在 Soot 开始加载待分析工程时，会将工程中的所有 Class 转换为 Soot 自定义的 SootClass，同时按照表4.2的对应关系，将每个 Class 中包含的子结构逐一封装成 Soot 中的相应类型对象。在 Soot 加载完毕后，逐一遍历 Scene 下所有的 SootClass，并分析其中的 Jimple 代码，在遇到调用语句解析其调用对象，即可构建 $Caller(U, S_m)$ 和 $Callee(M, S_u)$ 数据集合。具体构建过程如算法1描述。

以本节上面给出的代码片段为例，在对 foo 方法的 $Units$ 进行遍历时，在第 6 行遇到方法调用语句，可以得到其调用类型为 $virtualinvoke$。然后从该 $unit$ 中解析出被调用的对象 $r1$，这里无法得到 $r1$ 的类型，需要对 $r1$ 进行逆向分析，即 $backAnalysis$ 方法，直到解析到它是由该方法的第一个参数传递进来，从而得到它的类型为 $java.lang.String$。然后调用 $getMethod(className, methodName, invokeType)$ 方法，该方法传入的参数分别为 $r1$ 的类型的名称，被调用方法名字及调用方式，$getMethod$ 方法可以获得被调用方法在 Soot 环境下的实例。然后将 $unit$ 和 $Method$ 实例的映射信息加入到 $Caller$ 和 $Callee$ 这两个数据结构中。

Algorithm 1: 过程间调用图构建算法

Input: Soot 加载的 Class 集合 S_c
Output: 过程间调用图 G

$Caller \leftarrow \emptyset$
$Callee \leftarrow \emptyset$
foreach $c \in S_c$ **do**
  foreach $m \in c$ **do**
    foreach $u \in m.units$ **do**
      if $uinstanceofinvoke$ **then**
        $r = u$ 中执行调用的对象变量
        $mehodName = u$ 中被调用的方法的名字
      $className = backAnalysis(r)$
      if $u.invokeinstanceofinvokespecial$ **then**
        $invokeType = invokespecial$
      else if $u.invokeinstanceofinvokestatic$ **then**
        $invokeType = static$
      else if $u.invokeinstanceofinvokevirtual$ **then**
        $invokeType = invokevirual$
      else if $u.invokeinstanceofinvokeinterface$ **then**
        $invokeType = invokeinterface$
      $m' = getMethod(className, methodName, invokeType)$
      $Caller \leftarrow Caller \cup \{(u, m')\}$
      $Callee \leftarrow Callee \cup \{(m', u)\}$
$G \leftarrow \{\{Caller\}, \{Callee\}\}$
return G

本文需要将一个程序的结构刻画出来，只用方法内的控制流图和过程间的调用图是不够的，为了形象地反映 null 值在整个程序的传播路径，需要将空指针引用缺陷产生的上下文所涉及的方法构建成全局控制流图。在一个方法的调用语句处，将被调用方法的控制流图拼接进来，最终得到的全局控制流图将由若干个 unit 结点组成，结点之间的边不仅表示方法内语句的跳转关系，一些边还表示方法间的调用关系。用一张图表示整个程序的控制流结构有利于直观地反映整个程序的结构和复杂程度，也有利于代码特征的抽取和程序之间差异性的比较。

```
[21] public static void main(String[] args) {
[22]         ClassA varA = new ClassA();
[23]         Object object = null;
[24]         int varB = varA.foo();
[25]         if (varB == 0) {
[26]             object = new Object();
[27]         } else {
[28]             System.out.println("do nothing");
[29]         }
[30]         System.out.println(object.hashCode());
[31] }
```

```
[12] private int foo(){
[13]         return 1;
[14] }
```

上面的两个代码片段分别包含了 main 方法和 foo 方法的实现，在 main 方法的第 23 行处，*object* 对象被赋值为 null，在第 30 处发生了对 *object* 变量的解引用，此时必然会触发空指针引用缺陷。根据前文介绍，首先需要提取出分析域的范围，即第 23 行代码到第 30 行代码之间的控制流信息。同时，在 main 方法的第 24 行处存在对 foo 方法的调用，因此分析域应该包含 foo 方法内的控制流图。前文已经介绍了过程间调用图的构建方法，在过程间调用图的支持下，只需要在 main 方法的第 23 行处将 foo 方法内的控制流结构拼接起来就可以生成该分析域的全局控制流图，如图4.6所示。同理，即使方法的嵌套调用层级很多，也一样可以构建出分析域的全局控制流图。

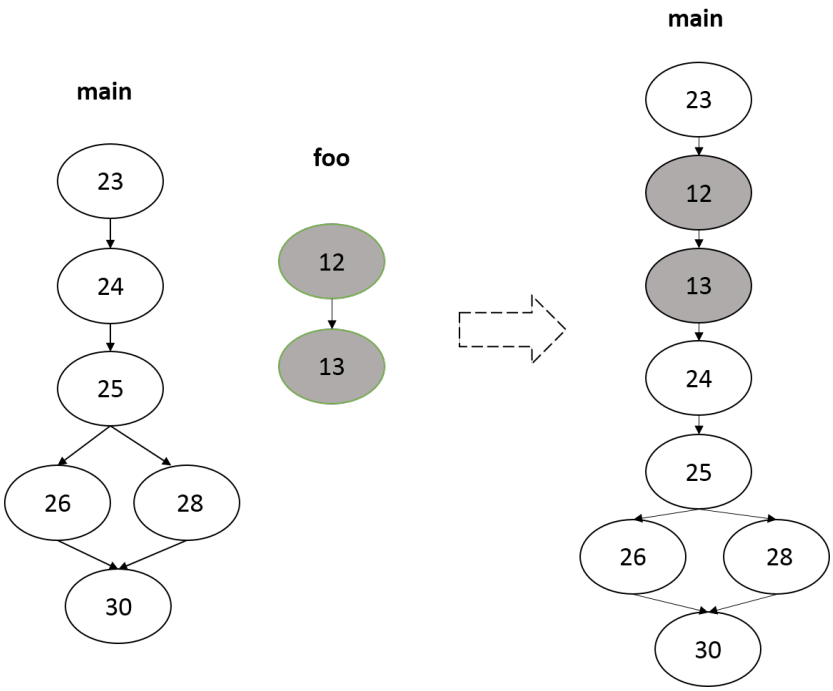


图 4.6　全局控制流图构建示例

4.3　代码特征抽取

分析域的全局控制流图可以表示空指针引用缺陷相关上下文代码的结构信息。但是代码不仅包含结构信息，还包含语义信息。由于本文构建的全局控制流图的结点为 Jimple 表示的 Unit，它表示了一个抽象的语法层面的语句类型，以 Unit 为基本单位。基于 Unit 的特点，可以选取一些维度的信息进行编码作为该结点的特征附加在控制流图结构上，这些维度分别如下：

1. 前驱结点数量

以构建的全局控制流图为基础，针对图中的每个结点获取其前驱结点的数量，作为该维度的特征值。

2. 语句类型

Unit 表示抽象的语句类型，实际上它包含了 15 种具体的 Stmt 语句类型，这些具体的语句类型和编码如表4.3所示。

3. 调用语句类型

调用语句表明该程序涉及到了跨过程的分析，在 Jimple 包含的 Stmt 中，实现 InvokeStmt 接口的类有五种，它们表示五种调用方法的类型，它们的编码如表4.4所示，如果该结点不包含方法调用语句，编码为 0。

表 4.3　Jimple 中 Stmt 语句类型及特征编码

语句类别	Stmt 名称	编码
核心指令	NopStmt	1
	IdentityStmt	2
	AssignStmt	3
方法内控制流指令	IfStmt	4
	GotoStt	5
	BreakPointStmt	6
	TableSwitchStmt	7
	LookUpSwitchStmt	8
方法间控制流指令	InvokeStmt	9
	ReturnStmt	10
	ReturnVoidStmt	11
监视器指令	EnterMonitorStmt	12
	ExitMonitorStmt	13
处理异常指令	ThrowStmt	14
	RetStmt	15

表 4.4　Jimple 中调用语句的类型及特征编码

调用指令	说明	编码
invokestatic	调用静态方法	1
invokespecial	调用私有方法、父类方法、构造方法	2
invokevirtual	调用抽象方法	3
invokeinterface	调用接口方法	4
invokedynamic	调用动态方法	5

4. 使用的操作数数量

该维度特征表示了指令对操作数使用的密集程度，过多的操作数使用往往也代表着算术指令的密集使用。

5. 空指针传递情况

如果当前节点涉及到了 null 值的传递操作，可以说明下文中有更多可能会触发空指针引用异常。空指针的传递也有不同的方式需要编码，如表4.5所示，如果没有发生空指针传递，编码为 0。

6. 空指针检查情况

空指针检查可以很好的避免下文中对空指针的引用，该语句通常会影响到工具检测结果的判定。如果该结点对 null 值进行了检查，编码为 1，否则为 0;

表 4.5　Jimple 中空指针传递类型及特征编码

空指针传递指令	传递方式	编码
ReturnStmt	方法返回值	1
InvokeStmt	方法调用传参	2
definitionStmt	直接赋值	3

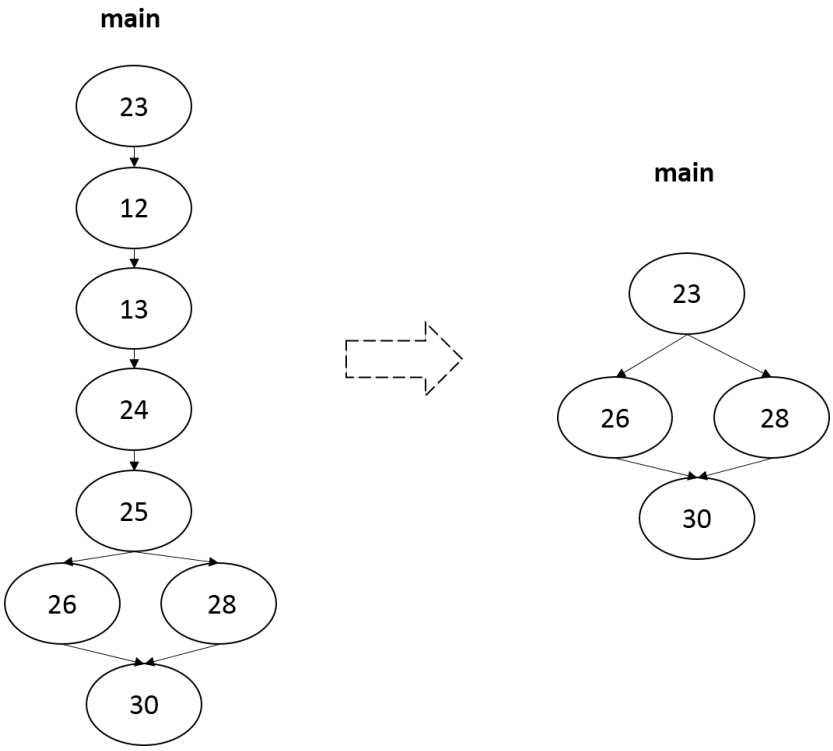


图 4.7　全局控制流图压缩示例

通过这六个维度的特征抽取，不仅可以体现全局控制流图的结构特征，而且可以体现图中每个结点即 Jimple 基本语句的语义特征。但是 Java 代码在转换为 Jimple 中间表示的过程中，为了使得到的语法单位都是标准的三地址表达式，将 Java 基本语句进行了大量的拆解重组，使得 Jimple 的基本语句的粒度比 Java 基本语句的粒度要小很多，这就使得生成的全局控制流图所能体现的特征更加不明显。为了在模型训练的过程中更加容易的对代码分类，需要将生成的全局控制流图进一步压缩，使得图中每个结点接近基本块的维度。具体操作如算法2。

经过此算法处理，图4.6生成的全局控制流图将被进一步压缩，如图4.7所示。此算法将图中的串行结点，即前驱和后继结点数量均为 1 的结点合并为 1 个结点。同时，在算法的第 16 行进行了 $mergeAttribute$ 的操作，这个操作可以将多个结点的特征属性转换到压缩后的一个结点上。具体转换方法如表4.6所示, 压缩后的图中结点的

Algorithm 2: 控制流图压缩算法

Input: 控制流图 G_c
Output: 压缩后的控制流图 G'_c

```
1  workList ← ∅
2  workList ← {G_c.RootNode}
3  while worklist ≠ ∅ do
4      node ← workList.next
5      remove node from workList
6      node set visited
7      if size of node.succList = 0 then
8          node set visited
9          continue
10     else if size of node.succList = 1 then
11         tmpNode ← node
12         while size of node.succList = 1 do
13             succNode ← node.succlist[0]
14             if size of succNode.succList > 1 OR size of succNode.precList > 1 then
15                 break
16             mergeAttribute(tmpNode)
17             tmpNode ← succNode
18             tmpNode set visited
19         node.succList ← tmpNode.succList
20         insert node into G'_c
21         foreach succNode in node.succList do
22             if succNode is not visited then
23                 insert succNode into workList
24     else
25         foreach succNode in node.succList do
26             if succNode is not visited then
27                 insert succNode into workList
28     insert node into G'_c
29 return G'_c
```

特征数量由压缩前的 6 个增加到了 8 个，这些特征取值由压缩前多个结点的特征综合得到。

表 4.6　控制流图压缩后的结点特征转换

压缩前特征值	压缩后特征值
前驱结点数量	压缩后结点的前驱结点数量
语句类型	方法内控制流指令数量 方法间控制流指令数量 其他指令类型数量
调用语句类型	被压缩结点中出现调用语句的数量
使用的操作数数量	被压缩结点中使用的操作数数量之和
空指针传递类型	被压缩结点中空指针传递次数
空指针检查情况	被压缩结点中空指针检查次数

4.4　数据标注

前文已经提到，本文训练模型的目的是为了评估不同工具对特定缺陷检测的能力，从而根据实际结果可以预测缺陷真实的可能性。因此本文训练数据的标签应该体现工具检测相应缺陷的能力，该能力记为置信度 $V(T,C,L)$，其中 T 表示工具类型,C 表示测试用例,L 表示测试用例中的某个具体位置，如果 $V(T,C,L)=1$，表示工具 T 在用例 C 的位置 L 处的检测结果是可信的，如果 $V(T,C,L)=-1$，则表示工具 T 在用例 C 的位置 L 处的检测结果不可信。

在本章第一节构建的数据集中包含两类用例，一部分用例中包含缺陷，另一部分则为代码结构与其相似但无缺陷的用例（生成缺陷用例的原始程序）。这些用例构建完成后会添加记录缺陷位置的注解，利用注解可以构建数据集的缺陷信息库。当测试用例 C 在位置 L 处存在空指针引用缺陷时，记为 $D(C,L)=1$，无缺陷时记为 $D(C,L)=0$。

分别使用每种工具对目标用例进行检测，并记录测试结果。如果工具 T 在测试用例 C 的位置 L 处检测出空指针引用缺陷，记为 $E(T,C,L)=1$，如果在该位置没有检测出空指针引用缺陷，则记为 $E(T,C,L)=0$。然后将工具检测的结果与缺陷信息库的结果进行比对，如果工具检出的空指针引用缺陷信息与缺陷信息库中相应的缺陷信息一致，即 $E(T,C,L)=D(C,L)$，则表示测试工具具备对该缺陷的检测能力，可得

置信度为 $V(T,C,L)=1$，否则表示测试工具不具备对该缺陷的检测能力，可得置信度 $V(T,C,L)=-1$。将该置信度的值作为用例数据的标签即可。

4.5 本章小结

本章介绍了训练模型需要的数据集的来源和构造方式，以及测试用例的控制流图构建和代码特征提取的方式，此外还介绍了利用图压缩的方式来提高特征辨识度从而提升训练效率的方法。最后介绍了训练数据的标签的含义。

第 5 章　深度学习模型的构建

本章节设计了一个根据一定的分类标准分类代码的神经网络模型。模型分为两部分：图结构特征抽取模型以及特征向量分类模型。图结构特征抽取模型将代码控制流压缩为 $1 * d$ 维的向量，特征向量分类模型根据训练数据的标签对特征向量进行分类。

模型以代码片段的包含节点信息的控制流图 (ACFG) g 为输入，在图结构特征抽取模型中，寻找一个合适的核函数 $\phi(\cdot)$ 将控制流图映射到高维空间，既能利用结点的特征信息，也保留了图结构信息。参考 Hanjun Dai[31] 等提出的方法，本文对控制流图数据用马尔可夫随机场进行建模，采用平均场推断的算法对 ϕ 函数进行估计，得到了 ϕ 的函数表达式。由于 ϕ 函数的具体参数未知，本文采用神经网络逼近该函数，并且将特征抽取模型与分类模型连接起来，通过对训练数据的学习估计 ϕ 函数。

在分类模型中，本文定义了一个具有一个隐含层的神经网络，为了避免深层的网络导致过拟合的情况，采用了 drop out 的方法随机丢弃了一些神经元的连接。分类模型将特征抽取模型中得到的特征向量作为输入，输出一个包含工具检测正确概率以及错误概率的二维向量 $[a, b]$。$a > b$ 时代表工具检测正确的概率较高；$a < b$ 代表工具检测错误的概率较高。当分类模型训练正确时，对一段陌生代码，模型应该能够正确评估不同工具对这段代码检测的结果是否正确。

在训练时，将两个模型作为一个整体，随机初始化网络中的权值，运用梯度下降以及后向传播 (Error BackPropagation, BP) 的方法，不断更新网络中的权值，直到模型预测的检测结果与实际的检测结果相接近为止。

5.1　图结构的数据向量化

5.1.1　核函数

一般来说，对于一个给定的训练数据集 $T = (x_1, y_1), (x_2, y_2), ...(x_n, y_n)$, 其中实例 x_i 属于输入空间，$x_i \in \chi = R^n$，对应两类标签 $y_i \in \gamma = -1, +1$，如果能用 R^n 中的一个超曲面将正负例正确的区分开，那么这个训练集则为线性不可分的。对于线性不可分的数据集，需要将原始空间映射到一个更高维的特征空间，使得数据集可以在这个特征空间里线性可分。一般来说，如果原始空间的数据维度是有限的，那么一定存在一个高维特征空间使样本可分。

令 $\phi(x)$ 表示将 x 映射过后的特征向量，那么在该特征空间下划分超平面的对应模型可以表示为：

$$f(x) = \omega^T \phi(x) + b$$

其中 w 和 b 是模型的参数。为了求出最优的划分超平面，需要求解以下最优化问题：

$$\min \frac{1}{2}||\omega||^2 \tag{5.1}$$

$$s.t. y_i(\omega^T \phi(x_i) + b) >= 1, i = 1, 2, ..., m. \tag{5.2}$$

对偶问题为：

$$\max \sum_{i=1}^{m} \alpha - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \phi(x_i)^T \phi x_j \tag{5.3}$$

$$s.t. \sum_{i=1}^{m} \alpha_i y_i = 0, \tag{5.4}$$

$$\alpha_i >= 0, i = 1, 2, ..., m \tag{5.5}$$

求解该优化问题需要计算 $\phi(x_i)^T \phi x_j$，该值为 x_i 和 x_j 映射到高维空间后的内积。由于高维空间的维度可能很高，甚至为无数维，因此直接计算该值是十分困难的，为了避免直接计算该值，可以定义一个核函数：

$$\kappa(x_i, x_j) =< \phi(x_i), \phi(x_j) >= \phi(x_i)^T \phi(x_j) \tag{5.6}$$

即 $\phi(x_i)^T \phi x_j$ 的值等于 $\kappa(x_i, x_j)$，有了核函数之后，就可以避免直接计算高维甚至无穷维的内积乘法。基于此可以重写对偶问题：

$$\max \sum_{i=1}^{m} \alpha - \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha_i \alpha_j y_i y_j \kappa(x_i, x_j) \tag{5.7}$$

$$s.t. \sum_{i=1}^{m} \alpha_i y_i = 0, \tag{5.8}$$

$$\alpha_i >= 0, i = 1, 2, ..., m \tag{5.9}$$

求解后得到分类函数：

$$f(x) = \sum_{i=1}^{m} \alpha_i y_i \kappa(x_i, x_j) + b \tag{5.10}$$

显而易见，当已知合适的映射函数 $\phi(\cdot)$ 的具体形式时，就可以写出核函数 $\kappa(\cdot,\cdot)$ 的具体形式。但是在现实问题中求解合适的映射函数是十分困难的。对于图结构的数据，由于其特殊性，需要同时考虑图中结点上的语义特征以及图本身的结构特征，因此，求解其映射函数需要特殊的方法，下文将介绍图结构数据常用的核函数。

5.1.2 图结构数据的核函数

对于结构化的数据，可以通过对序列子序列的计数来构建核函数。Leslie[37] 等人基于此提出了谱核函数 (Spectrum kernel) 的概念。具体来说，对于两个结构化的数据如字符串 χ 和 χ'，谱核函数定义为：

$$\kappa(\chi, \chi') = \sum_{s \in S} \#(s \in \chi)\#(s \in \chi') \tag{5.11}$$

其中 S 为 χ 和 χ' 序列中的所有可能的子序列的集合，$\#(s \in \chi)$ 函数统计子序列 s 在序列 χ 中出现的次数。相应的可以得到结构化数据的映射函数 $\phi(\chi) = (\#(s_1 \in \chi), \#(s_2 \in \chi), ...)^T$。类似的，Sherveashidez[38] 等人提出了图核函数 (graphlet kernel)，该函数用图结构的子图代替了谱核函数的子序列。

也可以运用概率图模型的方法来构建核函数。Jaakkola 和 Haussler[39] 提出了费舍尔核函数 (fisher kernel)，它定义了一个参数模型 $p(\chi|\theta*)$，并且通过极大似然估计的方法估计 $\theta*$，得到核函数的形式为：

$$\kappa(\chi, \chi') = U_\chi^T I^{-1} U_{\chi'} \tag{5.12}$$

其中 $U_\chi := \nabla_{\theta=\theta^*} \log p(\chi|\theta)$，$I = E_g[U_g U_g^T]$ 为费舍尔信息矩阵。

为了尽可能保存图结构数据的信息，同时对图结构进行高维的转换，从而进行图结构的分类。需要找到一个映射函数 ϕ，对图结构的数据 X，得到 $\mu_x = \phi(X)$，对于合适的核函数，得到的结果 μ 应该与 X 是等价的，也就是说对 μ 和 X 做相同的操作，两者的结果应该相同，即:

$$f(x) = \tilde{f}(\mu_x)$$

其中 f 和 $\tilde{f}$ 为等价的操作。在后文中将讲解如何求解 μ_x。

5.1.3 希尔伯特空间

希尔伯特空间将原有空间的概率分布映射到一个隐含的有限维的特征分布空间中 (Smola[40]):

$$\mu_X = E_x[\phi(X)] = \int_X \phi(x) * p(x)dx : P \mapsto F$$

其中 P 和 F 分别为原空间与映射后的空间。这样就将求特征函数 $\phi(X)$ 转化成了求 $\phi(X)$ 的期望，简化了求解过程。

5.2 图结构数据的建模

5.2.1 马尔可夫随机场

马尔可夫随机场 (Markov random field, MRF) 是典型的马尔可夫网，是一种著名的无向图模型。图中的每个结点表示一个或一组变量，结点之间的边表示两个变量之间的依赖关系，马尔可夫随机场有一组势函数，这是定义在变量子集上的非负实函数，用于定义场中的概率分布。

在马尔可夫随机场中，对于图中结点的一个子集，若其中任意两个结点都有边连接，则称该结点子集为一个“团”，若在一个团中加入另外任何一个结点都不能成“团”，则称其为“最大团”。如图5.1所示，x_1, x_2, x_3 不能构成“团”，x_2, x_5 为“团”而不是最大团。

在马尔可夫随机场中，多个变量之间的联合分布能基于团分解为多个因子的乘积，每个因子仅与一个“团”有关。对于 n 个变量 $x = x_1, x_2..., x_n$，所有团构成的集合为 C，与团 $Q \in C$ 对应的变量集合记为 x_Q，则联合概率 $P(x)$ 定义为:

$$P(x) = \frac{1}{Z} \prod_{Q \in C} \Psi_Q(X_Q)$$

其中 Ψ_Q 为团 Q 对应的势函数，$Z = \sum_x \prod_{Q \in C} \Psi_Q(X_Q)$ 为规范因子。

马尔可夫随机场具有两个重要的特性:

- 局部马尔可夫性：给定某变量的邻接变量，则该变量条件独立于其他变量。
- 成对马尔可夫性：给定所有其他变量，两个非邻接变量条件独立。

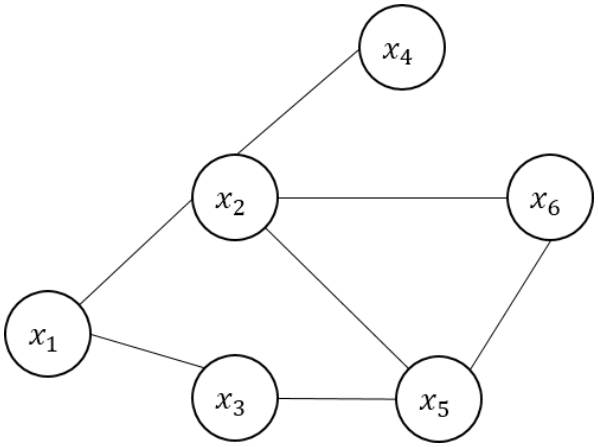


图 5.1　马尔可夫随机场

5.2.2　图结构数据的建模方法

Hanjun Dai[31] 等人用马尔可夫随机场对图结构数据进行了建模。首先定义一个图结构数据 χ，其结点集合为 $V=\{1,...,V\}$，边集合为 ξ。对图结点中的每一个结点 V_i，都有一个对应的特征向量 X_i。这里的特征向量 X 是能够被观测到的性质。相应的，对每一个特征向量 X_i，可以定义一个额外的隐变量 H_i，这样就将图结构的数据转换为了一个马尔可夫随机场，并且能够得到该随机场的联合概率分布:

$$p(\{H_i\},\{X_i\}) \propto \prod_{i\in V}\Phi(H_i,X_i)\prod_{(i,j)\in\xi}\Psi(H_i,H_j) \tag{5.13}$$

其中 Φ 为 H_i 到 X_i 的观测函数，Ψ 为 H_i 到 H_j 的状态转移函数。

对于这个模型，在生成隐变量的过程中，不仅将图上的每个结点上的特征作为输入，还考虑到了图的结构，将新生成的隐变量的结点按图结构连接起来，从而实现了同时保留结点的属性特征和图结构特征的目的。图5.2展示了建模的过程:

5.2.3　图结构数据模型的求解

直接求解以上的马尔可夫随机场是十分复杂的，需要对图结构中所有结点的相互关系进行考虑，然后求解，即:

$$p(H_i|x_i)=\int_{\mathcal{H}^{V-1}}p(H_i,\{h_j\}|\{x_j\}\prod_{j\in V})dh_i \tag{5.14}$$

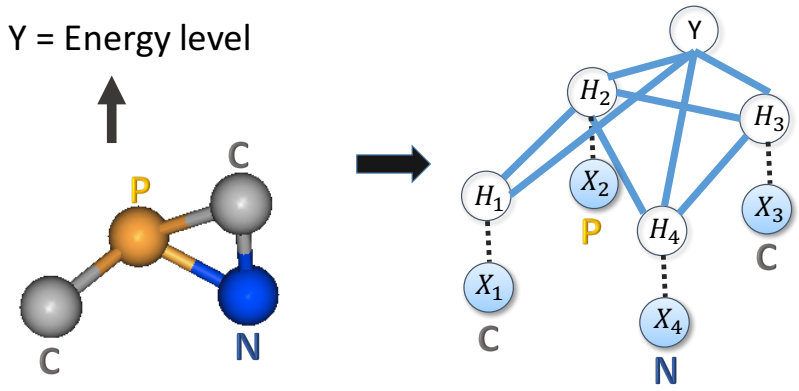


图 5.2　图数据隐变量建模

对上式的直接求解是十分困难的，现在主要的方法是运用平均场推断和信念传播网络求解估计值，本文采用了 Hanjun Dai[31] 提出的平均场推断的算法。

平均场推断算法尝试用一个独立的密度成分 $q_i(h_i)$ 来近似估计 $p(H_i|x_i)$，即 $p(H_i|x_i) = \prod_{i\in V} q_i(h_i)$，其中 $q_i(h_i) \geq 0$，在平均场中所有成分的概率和为 1 即 $\int_{\mathcal{H}} q_i(h_i)dh_i = 1$。求解平均场中密度成分需要最小化平均场中的自由能 (Wainwright 和 Jordan[41])，即求解以下的最优化方程：

$$\min_{q_1,\ldots,q_d} \int_{\mathcal{H}}^{d} \prod_{i\in V} q_i(h_i) \log \frac{\prod_{i\in V} q_i(h_i)}{p(\{h_i\}|\{x_i\})} \prod_{i\in V} dh_i \tag{5.15}$$

要实现上式的最小化，需要满足以下的等式：

$$\log q_i(h_i) = c_i + log(\Phi(h_i, x_i)) + \sum_{j\in\mathcal{N}(i)} \int_{\mathcal{H}} \log(\Psi(h_i, h_j)\Phi(h_j, x_j))dh_j \tag{5.16}$$

$$= c_i' + log(\Phi(h_i, x_i)) + \sum_{j\in\mathcal{N}(i)} \int_{\mathcal{H}} \log(\Psi(h_i, h_j))dh_j \tag{5.17}$$

可以得到：

$$c_i' = c_i + \sum_{j\in\mathcal{N}(i)} \int_{\mathcal{H}} \log \Psi(h_i, h_j)$$

其中 $\mathcal{N}(i)$ 为隐变量 $H(i)$ 在图模型中的相邻结点，c_i 为常数。从上式可以得知 $q_i(h_i)$

的值与 $h_i, x_i, \{q_j\}_{j\in\mathcal{N}(i)}$ 有关，由此可以定于一个关于 $q_i(h_i)$ 的等式：

$$q_i(h_i) = f(h_i, x_i, \{q_j\}_{j\in\mathcal{N}(i)}) \tag{5.18}$$

对于平均场中的每个组成结点 q_i，可以得到其在希尔伯特空间的映射:

$$\tilde{\mu}_i = \int_{\mathcal{H}} \phi(h_i) q_i(h_i) dh_i \tag{5.19}$$

已知 $q_i(h_i) = f(h_i, x_i, \{q_j\}_{j\in\mathcal{N}(i)})$，可以得到：

$$\tilde{\mu}_i = \boldsymbol{\tau}(x_i, \{\mu_j\}_{j\in\mathcal{N}(i)}) \tag{5.20}$$

其中 $\boldsymbol{\tau}$ 为 $\tilde{\mu}_i$ 与 $x_i, \{\mu_j\}_{j\in\mathcal{N}(i)}$ 的映射关系，这样不需要求解马尔可夫随机场中的 Φ, Ψ 两个函数，就可以得到 $\tilde{\mu}_i$ 的表示方法。对于 $\boldsymbol{\tau}$ 函数的具体函数形式可以通过对训练数据的学习来估计。本文采用了神经网络的方法来估计该函数。

5.3 分类模型的构建

5.3.1 神经网络

神经网络是由具有适应性的简单单元组成的广泛并行互联的网络，它的组织能够模拟生物神经系统对真实世界物体所作出的交互反应[42]。神经网络中最基本的成分是神经元模型，每个神经元与其他神经元互联，当它“兴奋”（接收到数据被激活) 时，就会向其相邻的神经元发送数据，改变相邻神经元的值，当相邻神经元的值超过某一阈值时，该神经元就被激活，从而向其他神经元继续发送数据。

1943 年，McCullich 和 Pitts[43] 将上述情形抽象为“M-P 神经元模型“，结构如图5.3所示。在这个模型中，神经元接受来自 n 个其他神经元的信号，这些信号通过带权重的连接进行传递，神经元接受到的总输入通过激活函数处理神经元产生的输出。

图5.4展示了常用的两种激活函数。理想中的激活函数为阶跃函数，当神经元激活时，输出 1，抑制时输出 0。然而在实际运用中，阶跃函数具有不连续，不光滑的性质。实际常用的函数有 Sigmoid 函数，可以把输入压缩到 (0,1) 的范围内，Relu 函数可以丢弃值为负的神经元。

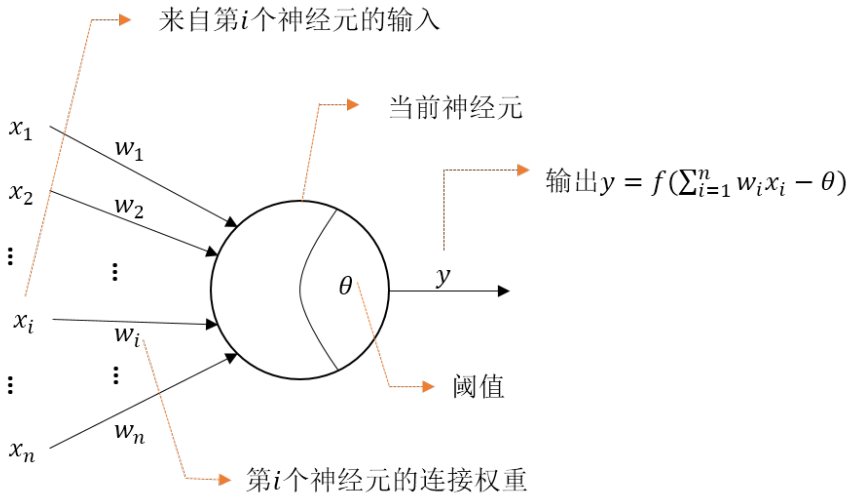
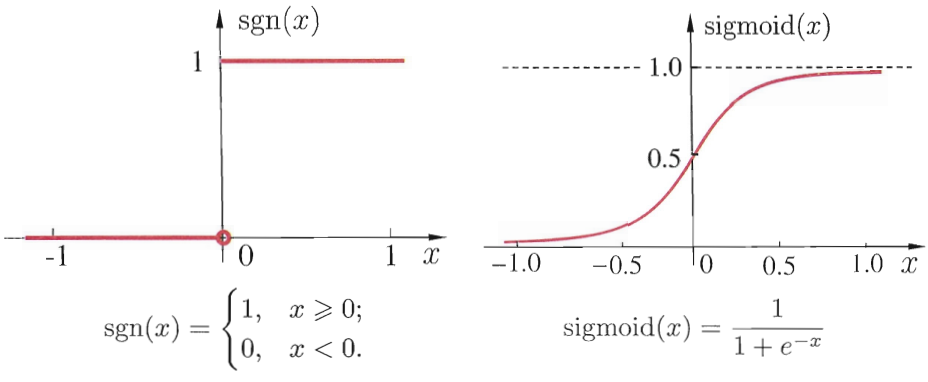


图 5.3　M-P 神经元模型



(a) 阶跃函数　　(b) Sigmoid 函数

图 5.4　常用激活函数

从数学的角度来看，神经网络实际上是一个包含了许多参数的数学模型，可以看作是一个由 $y_i = f(\sum_i w_i x_i - \theta_i)$ 这样的函数镶嵌叠加而成的复杂函数。当神经网络足够深时，这样的函数几乎逼近任意复杂度的连续函数，因此，可以用神经网络来代替难以求解的复杂函数，本文运用神经网络来逼近函数 $\boldsymbol{\tau}$。

5.3.2 图结构特征抽取模型构建

上文中可以得到 $\tilde{\mu}_i$ 的表示函数：

$$\tilde{\mu}_i = \boldsymbol{\tau}(x_i, \{\mu_j\}_{j \in \mathcal{N}(i)}) \tag{5.21}$$

先用一个一层的神经网络对该函数进行逼近，则：

$$\tilde{\mu}_i = \sigma(W_1 x_i + W2 \sum_{j \in \mathcal{N}(i)} \tilde{\mu}_j) \tag{5.22}$$

其中 σ 为 relu 的激活函数，W_1, W_2 为网络的权重。假设得到的隐变量 $\tilde{\mu}_i$ 的维数为 d，观测变量的维数为 p，那么可以知道，W_1 为一个 $p * d$ 的权值矩阵，W_2 为 $d * d$ 的权值矩阵。为了更好的逼近函数 $\boldsymbol{\tau}$，可以加深网络的深度 T，进行多轮迭代更新 $\tilde{\mu}_i$ 即：

$$\tilde{\mu}_i^t = \sigma(W_1 x_i + W2 \sum_{j \in \mathcal{N}(i)} \tilde{\mu}_j^{t-1}) \tag{5.23}$$

图5.5展示了其中一轮迭代的具体过程：对节点 X 的特征矩阵乘以权值矩阵 W_1，对 X 节点相邻节点的隐变量求和非线性化后加上 $W_1 * X$，再运用 $tanh$ 再次非线性化后得到下一轮迭代的初始值。

至此，对图结构数据中的每一个结点，都能够得到一个隐变量 $\tilde{\mu}_i$，为了将图压缩为一个 n 维的向量 g，可以将所有结点的隐变量求和:$g = \sum_{v \in V} \mu_v^{(T)}$。$g$ 中包含了图结构以及图结点的信息，对 g 进行分类相当于对图结构进行分类，并且 g 作为一个 n 维的向量，用神经网络继续分类将十分容易。得到 g 的具体过程如算法3所示：

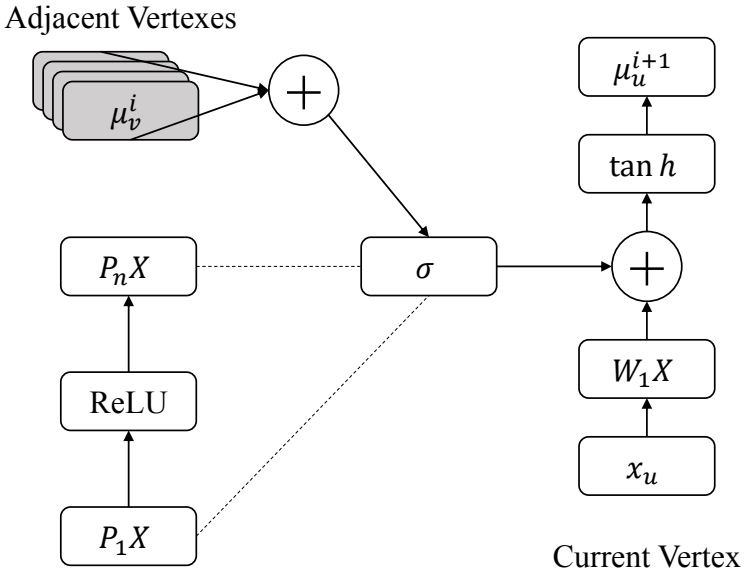


图 5.5　图特征抽取模型图示

Algorithm 3: Graph embedding algorithm

Input: $ACFGg = (V, \xi, x)$
Output: $\phi(g)$
Initialize $\mu_v^{(0)} = \overline{0}$, for all $v \in V$
for $t = 1$**to**T **do**
  foreach $v \in V$ **do**
    $l_v = \sum_{u \in \mathcal{N}(v)} \mu_u^{(t-1)}$
    $\mu_v^{(t)} = tanh(W_1 * x_v + \sigma(W_2 * l_v)$
return $\phi(g) = \sum_{v \in V} \mu_v^{(T)}$

5.3.3 判别模型的构建

明确了图数据压缩网络后，就可以通过训练数据来学习网络中的参数。为了判定某段代码能否被一个静态检测工具正确的检测，需要一些先验的数据集 $D = \{x_n, y_n\}_{n=1}^{N}$，其中 x_n 为抽取出来的代码的控制流图，y_n 为代码的标签。在第四章的数据标注中，当代码能正确被某工具检测，标签记为 1；不能被正确检测则标签记为-1。本文对该标签进行了"One Hot Encoder" 处理用于模型训练，即如果该代码能够被工具正确检测，$y_n = [1, 0]$，否则 $y_n = [0, 1]$，将问题转换为一个二分类的问题。为了解决该问题，需要学习得到一个网络，使网络得到的值能够最大的拟合训练数据，即：

$$\min \sum_{n=1}^{N} (y_n - P * \phi(g)) \tag{5.24}$$

$$= \min \sum_{n=1}^{N} y_n - P * (\sum_{i=1}^{V_n} \tilde{\mu}_i^n) \tag{5.25}$$

为了更好的拟合数据集 D，逼近真实函数，本文在图结构压缩网络上定义了一个两层的非全连接网络 P，P 在两层全连接网络的基础上随机丢弃了一些神经元，从而避免了全连接网络可能存在的过拟合问题，具体来说，P 可以定义为：

$$P(\phi(g)) = W_4 * (relu(W_3 * relu(\phi(g)))) \tag{5.26}$$

其中 W_3 为 $d * d$ 的权值矩阵，W_4 为 $d * 2$ 的权值矩阵。这样就得到了一个完整的图结构数据分类网络，该网络有四个权值矩阵 W_1, W_2, W_3, W_4，以代码的控制流图为输入，输出一个 $1 * 2$ 的向量 $[a, b]$。当输出 $a > b$ 时，代表工具检测正确的可能性大于错误的可能性；若 $a < b$ 代表工具检测错误的可能性更大。本文用梯度下降的方法对网络的权值进行更新，当输出值 $[a, b]$ 与训练集的标签 y_n 残差较小且趋于稳定时，就得到了一个较好的用于区分某段代码能否被工具正确检测的分类网络。

将图结构特征抽取模型与判别模型连接在一起，就得到了本文模型的完整架构：图特征抽取模型对图特征进行压缩，判别模型根据分类标准对压缩后的特征向量进行分类。网络的完整架构见图5.6。

基于此，对于四个静态代码分析工具，最终能训练出四个相应的分类器，这样，当检测目标代码时，四个分类器能够分别得到四个工具检测这段代码的置信度 $W =$

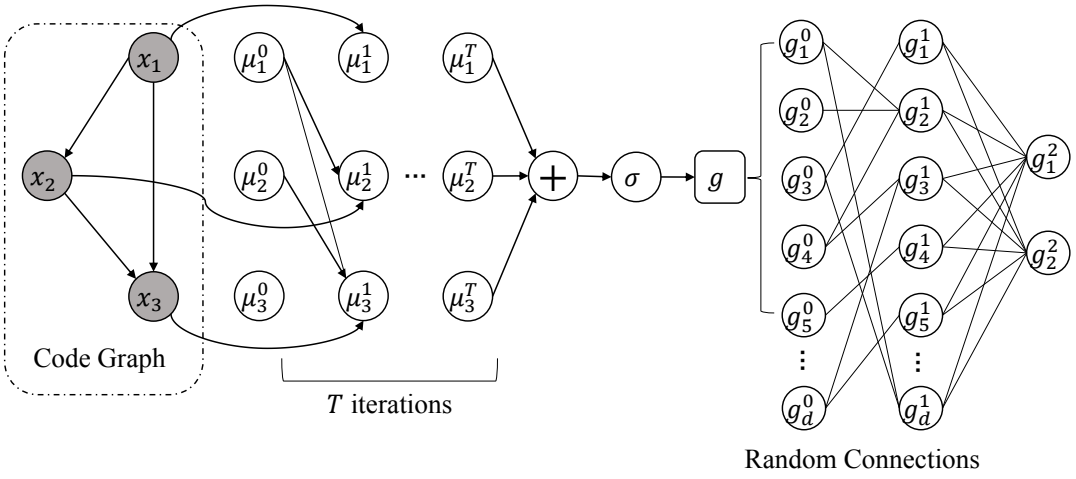


图 5.6　完整分类模型图示

$\{w_1, w_2, w_3, w_4\}$，它们的取值范围都是 [0-1]。同时还能够得到四个工具对这段代码的检测结果 $R = \{r_1, r_2, r_3, r_4\}$，这些结果的取值为 1 或-1。综合两个结果，就能够得到这段代码是否真正出现了空指针引用缺陷:

$$P = W^T * R = \sum_{i=1}^{4} w_i * r_i$$

设定好一个阈值 p，当 $P \leq p$ 时，推断该代码段存在空指针引用缺陷；$P > p$，推断该代码不存在空指针引用缺陷。阈值 p 也可以通过学习得到，通过试错的方法逐渐更新 p 值，直到在训练集和测试集上得到最高的测试准确率时，停止更新。

5.4　本章小结

本章介绍了代码分类神经网络模型的基本构造，阐述了图数据特征抽取模型的理论基础以及设计原理，介绍了分类模型神经网络的连接方式。最后阐明了神经网络的训练方法以及评判标准。

第 6 章　实验评估

6.1　实验环境

本次实验的环境为 Windows 10 64 位系统，CPU 为四核 Intel Xeon 3.3Ghz CPU，24G 内存。代码开发环境为 Python3.5 和 TensorfFow1.8.0 版本。

TensorFlow 是谷歌旗下的一款深度学习框架，是一个使用数据流图进行数值计算的开放源代码软件库。利用 TensorFlow 框架可以轻松实现深度学习模型而不需要考虑算法的底层实现，并且 TensorFlow 提供多个优化器可供选择，其灵活的参数设置机制让使用者能够快速调整深度学习模型。因为其方便性与易用性，TensorFlow 已经成为了机器学习工作者实现其模型的首选框架，这也是本文采用 TensorFlow 实现文中算法的原因。

6.2　数据选择

在第四章的数据集构建部分，总共生成了 12780 个用例。其中 5340 个用例为不含空指针引用缺陷的正常用例，7440 个样例为包含空指针引用缺陷的用例。由于对于无缺陷的用例，四个工具大都能正常鉴别，区分度较小，所以在训练数据时限制了这些用例的数量，只是从这些用例中随机抽取 1000 个作为训练数据。对于包含空指针引用缺陷的用例，则需要去除掉一些控制流图节点数量异常和特征矩阵数据过于稀疏的用例。最终本文选取的训练集用例数量为 8650 个，对于这些数据，各个工具的检测结果已经在第三章的相关背景中的表3.1中给出。其中，每个用例控制流图包含的结点数量分布如图6.1所示，包含的结点数目多少反映了用例程序的复杂程度。

如果直接将这批训练集放入模型训练的话，由于不同工具检测能力的差别，会导致标记的正负样本数量分布不均的情况，很容易出现模型直接收敛到样本多数值的问题，造成最终的模型泛化能力过低。为了解决样本分布不均的问题，本文试验了两种常用的方法，过采样和欠采样的方法。如图 6.2 所示，过采样方法随机重复采样数量较少的样本，从而达到正负样本数量均衡的效果，不过重复的采样容易导致过拟合的问题。欠采样的方法随机丢弃一些多数样本的数据，在训练模型时保持正负样本数据数量的一致。

经过对比，本文选择了欠采样的方法。例如针对 FindBugs 工具，训练时在 5834

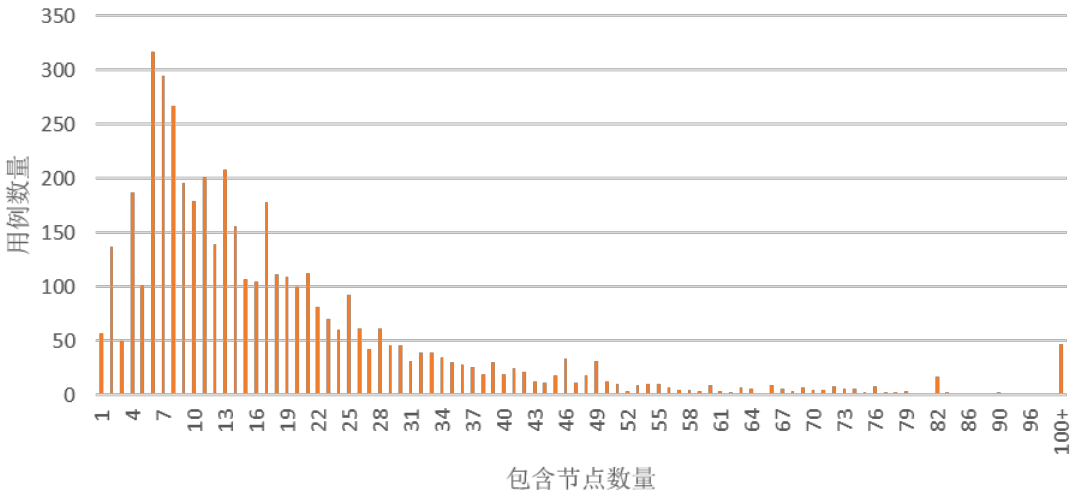
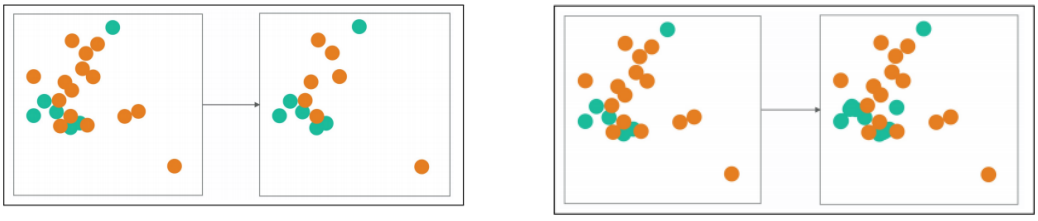


图 6.1　数据集中图结点数量的分布



(a) 欠采样方法示例　　(b) 过采样方法示例

图 6.2　两种采样方法示例

个检测正确的样本中随机挑选出 1488 个，与 1488 个缺陷样本一起加入训练，保证了训练数据正负样本的数量均衡。随机欠采样的方法既不会影响数据的统计特征，同时可以使正负样本的边界更为清晰。本文的测试数据采用了不同于训练数据的 1000 个代码样例，同样的，为了保证正负样本数量的均衡，每次测试时，本文从 1000 个样本中随机抽取正负样本各 50 个进行测试，测试十次得到测试的平均正确率。

6.3　训练优化实验

本次模型训练采用了 TensorFlow 的 AdamOptimizer 函数，AdamOptimizer 实现了 Adam 优化算法，比起随机梯度下降的方法，Adam 算法收敛速度更加迅速，陷入局部最优解的概率更小。当学习率过大时，容易导致模型发生振荡无法收敛，当学习率过小时，会导致模型收敛速度太慢耗时较长。为了避免上述情况的发生，本文采用了

动态学习率的方法。训练学习率初始设置为 10^4，学习率随着训练的迭代步数增加而呈阶梯状下降，每迭代 100 次降为原来的 0.98 倍。损失函数采用了交叉熵的算法。对两个概率分布 p 和 q，通过 p 来表示 q 的交叉熵为：

$$H(p,q) = -\sum_{x} p(x) \log q(x)$$

通过交叉熵函数可以比较两个概率分布的差异。交叉熵越小，表示两个概率分布差异越小，优化器可以根据交叉熵的值更新模型参数。

针对本文提出的模型，有两个参数可能影响模型的结果：图压缩后特征向量的维度大小 d 以及压缩算法的迭代次数 T。接下来将分别探讨两者的影响。

对于特征向量 d，可以看出，当 d 的维度增加，网络中的参数 W_1, W_2, W_3, W_4 的维度都会相应的增加，相当于网络的每一层的神经元都会增加。网络变得更加复杂，能够拟合的函数也就越复杂。与此同时，模型的训练难度和时间消耗也会增加。网络如果过于复杂也有可能导致模型过拟合的问题。

对于循环次数 T，相类似的，随着循环次数的增加，网络的层数逐渐增加，网络也会更加复杂，进而导致训练时长的增加和过拟合问题。

为了找到最佳的特征向量维度 d 以及迭代次数 T，本文在迭代次数 1 到 4 次，特征维度 8 到 512 维之间分别进行了实验。实验采用了相同的训练数据，当模型收敛后对相同的测试集进行测试，得到四个分类器的准确率均值，结果如图6.3。由图可知，迭代次数为 3，特征向量为 32 维时，训练效果最佳。

6.4 模型检测实验

在上文选择的训练集上进行训练之后，每一个工具都得到了一个二分类器，分类器在测试集上的分类准确率达到了 80% 以上。为了让实验结果更加清晰，本文对模型产生的高维数据进行可视化。首先运用训练好的图特征抽取模型对测试代码控制流图进行向量化得到图特征向量，然后运用 t-SNE 算法将特征向量压缩为一个二维向量，根据各个工具对该代码检测结果的正确与否给该数据标记不同的颜色，得到了图**??**的结果。

可以看出 FindBugs 的结果分布比较稀疏，说明了 FindBugs 的检测规则比较丰富，能够检测的代码类型较多；而其他三种工具的结果分布相比之下更聚集一些，说明其

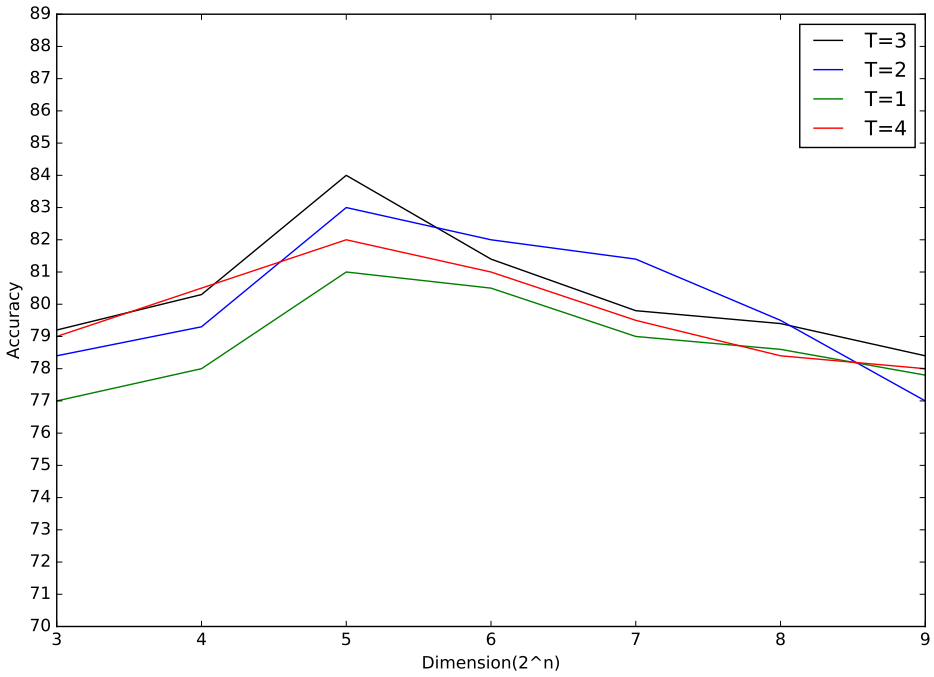
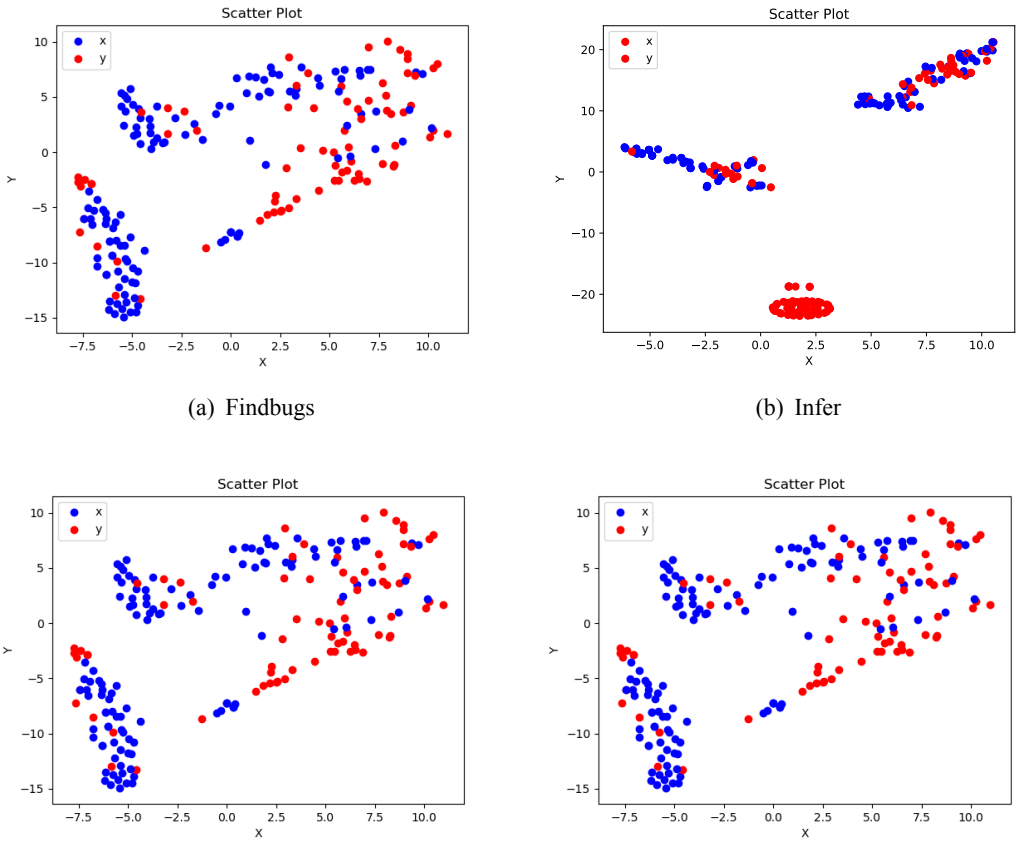


图 6.3　迭代次数和特征维度对模型训练的影响

检测所需的特征比较单一，以该工具检测能力为标准对代码的分类更加明显。

为了更清晰的展示训练好的分类器分类代码的准确率，本文从 1000 个训练集中随机抽取了 100 个样例绘制了 ROC 曲线，如图6.5所示。可以看到四个分类器都取得了一定的分类效果，FindBugs 分类器和 Infer 分类器的效果最好，Fortify 分类器的效果较差，这说明了 FindBugs 和 Infer 在检测空指针缺陷时，规则较为清晰，选取的代码特征比较明显，而 Fortify 采用分析数据流的方法检测空指针缺陷，对规则的依赖较少，因此分类较为模糊。

得到四个工具对应的分类器后，可以综合分类模型预测的结果和工具的实际检测结果对用例产生缺陷的真实性进行判定。为了标识工具实际的检测结果，这里对数据标签进行再处理，当工具对测试用例实际能够检测出空指针引用缺陷时，该数据的标签置为 1；当工具不能在该用例上检测出空指针缺陷时，标签置为-1。这样就可以得到四个工具在一个用例上的检测结果向量 $R = r_1, r_2, r_3, r_4$，同时模型可以得到四个工具检测结果的置信度矩阵 $W = w_1, w_2, w_3, w_4$。则测试用例含有空指针引用缺陷的置信度为 $P = W^T * R = \sum_{i=1}^{4} w_i * r_i$。这样得到的结果具有一定的容错性，当一个分类器出现预测错误时，其他三个分类器的结果可以弱化此分类器的错误，从而最小化单



(a) Findbugs (b) Infer

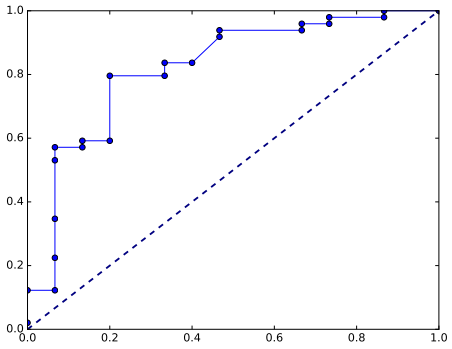
(c) Jlint (d) Fortify

图 6.4　四种工具的二分类模型的分类效果

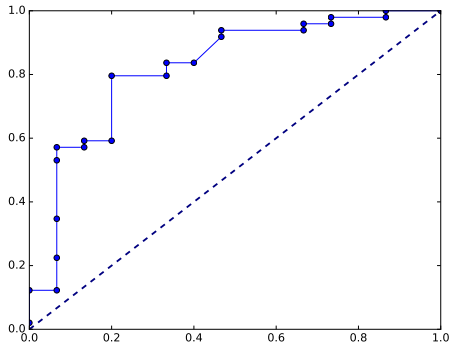
个分类器的分类错误对整个判定模型的影响。为了实现对目标代码是否具有缺陷的最终判定，需要设定一个阈值 p，当 $P >= p$ 时判定该代码存在空指针缺陷。

当模型分类完全没有误差时，阈值 p 应该接近于 0。然而在现实中模型无法做到没有偏差，为了得到 p 最佳的取值，本文从-0.3 到 0.4，每隔 0.05 取一个阈值进行实验。实验从测试集中随机选取了 1000 个用例，使用这些数据对模型进行检验，计算缺陷判定准确率后得到图6.6的结果：

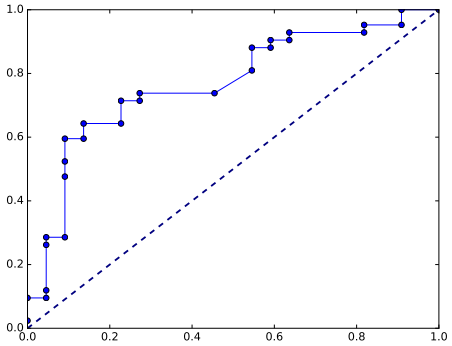
可以看出，当阈值为 0.05 时能够取得最大的检测准确率 81.34%。然后，使用该阈值下的测试结果和在这些数据上不同工具的检测结果进行对比，可得表6.1。表中的 BIT-Detector 表示使用模型对所有缺陷进行验证得到的结果，可以看出，模型在综合了四种工具的检测结果置信度后，成功对工具检测的结果进行了修正，提高了检测的准确率。表中的 BIT-Detector* 只选取四种静态检测工具检测一致的结果作为输出，该部分结果的准确率依然最高，但这是用较高的漏报代价达到的。



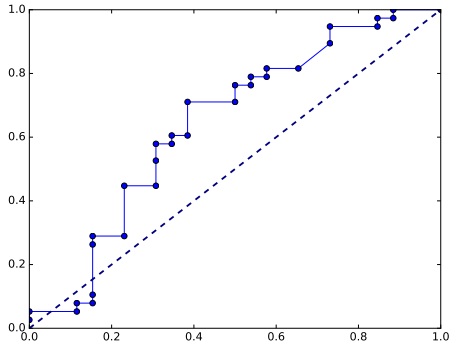
(a) Findbugs



(b) Infer



(c) Jlint



(d) Fortify

图 6.5　四种工具的二分类模型的 ROC 曲线

表 6.1　训练集上工具检测结果

工具	正确数量	错误数量	准确率
FindBugs	760	240	76.0%
Infer	648	352	64.8%
Jlint	634	366	63.4%
Fortify	727	273	72.7%
BIT-Detector*	378	54	87.5%
BIT-Detector	813	187	81.3%

虽然模型在工具已有检测结果的基础上利用代码分类可以给出更高的检测准确率，但是实验数据表明该准确率依然没有达到理想水平，不同工具能够同时检测的缺陷依然具有相当高的可信度。因此 BIT-Detector 使用可信度优先级排序的方式是最优解决方案，在不降低漏报率的前提下，报告靠前的部分拥有相对理想的可信度。后面的报告可以结合模型输出的缺陷置信度进行排序，这样排序的合理性能够得到保证，从而大幅提升检测报告的实用性。

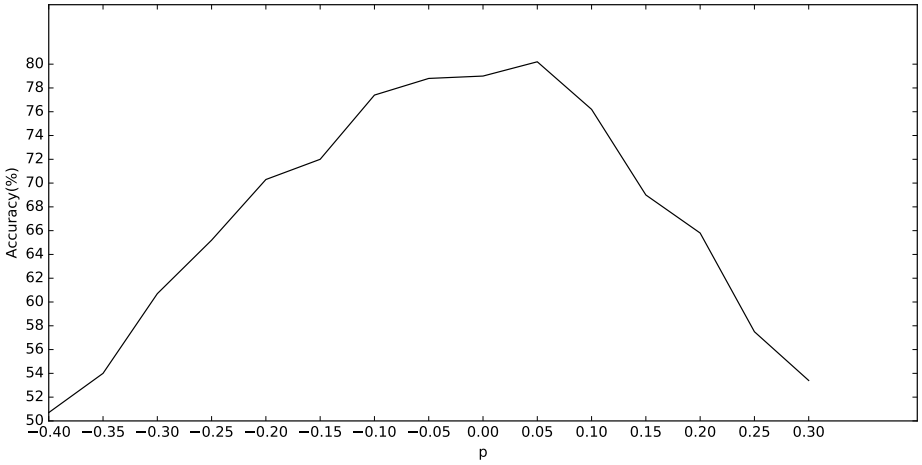


图 6.6　不同阈值下的缺陷判定准确率

6.5　本章小结

本章作为实验部分，首先对试验环境进行了介绍，并给出选择试验环境的原因。然后对模型训练和测试时选择的数据进行了介绍，随后对模型训练过程中的优化实验进行了分析，最后对模型的分类效果和最后预测效果进行了实验总结。

结论

随着软件规模的增长，代码安全问题日益突出。在众多的缺陷类型中，空指针引用缺陷出现的比例较高，且难以被检测，而其一旦出现将会带来巨大的损失。因此空指针引用缺陷的检测问题一直都是研究人员关注的热点。到目前为止，代码静态检测领域涌现出了大量的工具用以检测空指针引用等缺陷问题，但是研究表明这些工具的检测报告包含了大量的误报，导致了实用性的不足。而由于静态代码检测技术的限制，各种工具在权衡检测准确性和全面性方面做了不同取舍，因而导致不同工具产生的缺陷报告具有较大差别。

本文提出了一种交叉验证的思路，综合不同工具的检测报告来提升代码检测结果的精确率和覆盖率。并据此开发了基于 SonarQube 平台运行的插件——BIT-Detector。该插件可以在软件的持续集成环境下同时使用多种工具对代码进行检测，并且能够有效综合不同工具产出的缺陷报告，根据缺陷的可信度进行排序，从而提升单一工具在代码检测上的不足之处。其中，对于部分缺陷的置信度评估是本文研究的重点，本文利用深度学习方法，在构建了大量空指针引用缺陷用例的基础上，训练出图的向量化模型和依据工具检测能力的代码分类模型，其中进行了大量工作来建模缺陷代码的整体特征。

本文主要工作成果有下面四点：

(1) 提出交叉验证现有静态检测工具的思想，成功实现基于 SonarQube 平台的插件 BIT-Detector，该工具利用缺陷可信度优先级排序的方式可以大幅提升检测报告的实用性。

(2) 提出一种批量生成缺陷代码用例的方法，利用 AST 工具解析目标代码的抽象语法树，在合适的位置适当修改代码的语法结构，利用脚本自动化编译，执行并验证生成效果。依据此方法成功利用现有开源代码生成了大量空指针引用缺陷用例。

(3) 建模了空指针缺陷用例的整体特征。本文利用 Soot 工具构建空指针引用缺陷代码的控制流图，并结合过程间调用图生成程序的全局控制流图，继而对该图进行合理压缩，在基本块的层次上提取合理结点语义特征，使用包含语义特征的全局控制流图来表示一段空指针引用缺陷程序。

(4) 利用深度神经网络将代码的结构特征和语义特征向量化，并构建学习模型对缺陷代码依据工具的检测效果进行分类，从而评估多工具检测报告中出现的结果不一

致的缺陷的真伪性。

本文仅仅是探讨对 Java 代码中出现的空指针引用缺陷的检测，理论上本文使用的方法可以应用于任何代码缺陷和任意代码语言。未来的工作除了继续完善本工具的集成，性能和稳定性，也将会尝试扩展本文使用的方法，完成对更多缺陷类型的检测和更多语言的支持。

参考文献

[1] Chess B, Arkin B. Software security in practice [J]. IEEE Security & Privacy, 2011, 9 (2): 89–92.

[2] 单国栋, 戴英侠, 王航. 计算机漏洞分类研究 [J]. 计算机工程, 2002, 28 (10): 3–6.

[3] Vassilev A, Hall T A. The importance of entropy to information security [J]. Computer, 2014, 47 (2): 78–81.

[4] 李潇, 刘俊奇, 范明翔. WannaCry 勒索病毒预防及应对策略研究 [J]. 电脑知识与技术: 学术交流, 2017, 13 (7): 19–20.

[5] 张钢. [S. l.]: . [s. n.], 2014.

[6] Hovemeyer D, Spacco J, Pugh W. Evaluating and tuning a static analysis to find null pointer bugs [C]. In ACM SIGSOFT Software Engineering Notes, 2005: 13–19.

[7] 宫云战, 赵瑞莲, 张威. 软件测试教程 [M]. 2008.

[8] 宫云战. 软件测试 [M]. 国防工业出版, 2006.

[9] Coverity Scan Report. `https://www.synopsys.com/blogs/software-security/`.

[10] Osman H, Leuenberger M, Lungu M, et al. Tracking null checks in open-source Java systems [C]. In Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on, 2016: 304–313.

[11] Wei P. The Safety Of Software Design Loophole Dynamic State Examination Technique Analysis [J] [J]. CD Technology, 2009, 4: 015.

[12] Lin Z, Qing-kai Z. Static detecting techniques of software security flaws [J]. Computer Engineering, 2008, 34 (12): 157–159.

[13] King J C. Symbolic execution and program testing [J]. Communications of the ACM, 1976, 19 (7): 385–394.

[14] Bush W R, Pincus J D, Sielaff D J. A static analyzer for finding dynamic programming errors [J]. Software-Practice and Experience, 2000, 30 (7): 775–802.

[15] Jhala R, Majumdar R. Software model checking [J]. ACM Computing Surveys (CSUR), 2009, 41 (4): 21.

[16] Ball T, Rajamani S K. Automatically validating temporal safety properties of interfaces [C]. In Proceedings of the 8th international SPIN workshop on Model checking of software, 2001: 103–122.

[17] Tiwari A, Gulwani S. Logical interpretation: Static program analysis using theorem proving [C]. In International Conference on Automated Deduction, 2007: 147–166.

[18] Flanagan C, Leino K R M, Lillibridge M, et al. PLDI 2002: Extended static checking for Java [J]. ACM Sigplan Notices, 2013, 48 (4S): 22–33.

[19] Cousot P, Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints [C]. In Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1977: 238–252.

[20] Madhavan R, Komondoor R. Null dereference verification via over-approximated weakest pre-conditions analysis [J]. ACM Sigplan Notices, 2011, 46 (10): 1033–1052.

[21] Xie Y, Aiken A. Saturn: A scalable framework for error detection using boolean satisfiability [J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 2007, 29 (3): 16.

[22] 王锐强. 基于判断逻辑的空指针引用模式检测 [D]. [S. l.]: 北京邮电大学, 2015.

[23] 王旭. 基于控制流分析和数据流分析的 Java 程序静态检测方法的研究 [D]. [S. l.]: 西安电子科技大学, 2015.

[24] Loginov A, Yahav E, Chandra S, et al. Verifying dereference safety via expanding-scope analysis [C]. In Proceedings of the 2008 international symposium on Software testing and analysis, 2008: 213–224.

[25] Wala. http://wala.sourceforge.net.

[26] Nanda M G, Sinha S. Accurate interprocedural null-dereference analysis for Java [C]. In Proceedings of the 31st International Conference on Software Engineering, 2009: 133–143.

[27] 杨睿. 数组空指针故障的静态测试方法与实现 [D]. [S. l.]: 北京邮电大学, 2012.

[28] 姜淑娟, 王兴亚, 张艳梅, et al. 空指针异常的自动故障定位方法 [J]. 通信学报, 2017, 36 (1): 18–29.

[29] Hovemeyer D, Pugh W. Finding more null pointer bugs, but not too many [C]. In Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, 2007: 9–14.

[30] White M, Tufano M, Vendome C, et al. Deep learning code fragments for code clone detection [C]. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016: 87–98.

[31] Dai H, Dai B, Song L. Discriminative embeddings of latent variable models for structured data [C]. In International Conference on Machine Learning, 2016: 2702–2711.

[32] Xu X, Liu C, Feng Q, et al. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection [J/OL]. CoRR, 2017, abs/1708.06525. http://arxiv.org/abs/1708.06525.

[33] Rutar N, Almazan C B, Foster J S. A comparison of bug finding tools for Java [C]. In Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on, 2004: 245–256.

[34] Owasp. https://www.owasp.org/index.php/Main_Page.

[35] NIST. https://www.nist.gov/.

[36] Vallée-Rai R, Co P, Gagnon E, et al. Soot: A Java bytecode optimization framework [C]. In CASCON First Decade High Impact Papers, 2010: 214–224.

[37] Leslie C, Eskin E, Noble W S. The spectrum kernel: A string kernel for SVM protein classification [M] // Leslie C, Eskin E, Noble W S. Biocomputing 2002. World Scientific, 2001: 2001: 564–575.

[38] Shervashidze N, Vishwanathan S, Petri T, et al. Efficient graphlet kernels for large graph comparison [C]. In Artificial Intelligence and Statistics, 2009: 488–495.

[39] Jaakkola T S, Diekhans M, Haussler D. Using the Fisher kernel method to detect remote protein homologies. [C]. In ISMB, 1999: 149–158.

[40] Smola A, Gretton A, Song L, et al. A Hilbert space embedding for distributions [C]. In International Conference on Algorithmic Learning Theory, 2007: 13–31.

[41] Wainwright M J, Jordan M I, et al. Graphical models, exponential families, and variational inference [J]. Foundations and Trends® in Machine Learning, 2008, 1 (1–2): 1–305.

[42] Kohonen T. An introduction to neural computing [J]. Neural networks, 1988, 1 (1): 3–16.

[43] McCulloch W S, Pitts W. A logical calculus of the ideas immanent in nervous activity [J]. The bulletin of mathematical biophysics, 1943, 5 (4): 115–133.

攻读学位期间发表论文与研究成果清单