南京大学

研 究 生 毕 业 论 文

（申请工学硕士学位）

论 文 题 目	基于多维安卓特征的移动应用风险评估技术
作 者 姓 名	龚爱
学科、专业名称	工学硕士（软件工程领域）
研 究 方 向	软件工程
指 导 教 师	陈振宇 教授

2021 年 5 月 13 日

学　　号：**MG1832002**

论文答辩日期：**2021 年 5 月 13 日**

指 导 教 师：　　**（签字）**

Risk Assessment Technology for Mobile Application Based on Multi-Dimensional Android Feature

By

Ai Gong

Supervised by

Professor **Zhenyu Chen**

A Thesis
Submitted to the Software Institute
and the Graduate School
of Nanjing University
in Partial Fulfillment of the Requirements
for the Degree of
Master of Engineering

Software Institute

May 2021

南京大学研究生毕业论文中文摘要首页用纸

毕业论文题目：基于多维安卓特征的移动应用风险评估技术

工学硕士（软件工程领域）专业 2018 级硕士生姓名：龚爱

指导教师（姓名、职称）：陈振宇 教授

摘 要

随着安卓应用程序在日常生活中的广泛应用，应用程序的安全性和质量保障变得越来越重要。为了评估应用程序的潜在风险，人们引入了 Java 静态代码指标来预测应用中的缺陷。但是现有技术忽略了安卓应用的自身特性，且预测结果对缺陷代码的修正无指导意义，导致无法定位并修复应用中的风险漏洞。

本文提出了一种基于多维安卓特征的移动应用风险评估技术，用于检测应用质量和安全风险。其中质量风险指代了故障倾向性，涵盖了安全，内存，性能等各项风险问题。我们总结了安卓应用程序中的不良代码结构，定义了 15 种全新的代码异味指标，并针对性的给出了代码修复意见。为了能自动化检测出这些代码异味，我们开发了一个名叫 DACS 的检测工具。从编程规范维度出发，我们从源码中提取出 15 种自定义的代码异味和 15 种公开定义的代码异味。从规模，复杂度，重复和问题四个维度出发，我们提取出 21 种 Java 静态代码指标。在对这些特征指标预处理后，我们应用 9 种机器学习算法，构建了应用程序的安全风险等级预测模型，帮助开发者识别出高风险的应用。此外，我们选取了 5 种代码异味和代码行数指标作为自变量，应用程序的故障数作为响应变量，应用 2 种离散回归算法，构建了故障数模型，研究了代码异味对应用程序质量的影响。

为了验证 DACS 工具的有效性，我们在 20 个开源应用程序上运行了该工具，得到了一份检测结果。同时，我们人工检测了这些应用程序，分析该结果和工具检测结果之间的一致性关系。结果表明 DACS 工具和人工检测结果保持一致，可以代替人工检测。为了评估风险等级预测模型，我们从 GitHub 上收集了 4575 个安卓应用程序进行了三个实验。结果发现：（1）安卓代码异味有助于提高模型的风险预测性能（2）随机森林算法所建立的预测模型性能最好（AUC=0.97）（3）非静态内部方法（MIM）和内部类泄漏（LIC）等代码异味容易造成程序的高风险，开发人员应予以重视。此外，我们还在 645 个应用程序上，构建了故障数模型。结果表明，与负二项回归模型相比，零膨胀负二项回归算法构建的计数模型更优（AIC=517.32，BIC=522.12）。代码行数 (NCLOC)，恶意压缩（MU），弱加密算法和代码行数的组合指标（WCA:NCLOC）会显著增加应用程序的质量风险。

关键词： Android 代码异味，Java 静态代码指标，Android 风险，故障数

南京大学研究生毕业论文英文摘要首页用纸

THESIS: Risk Assessment Technology for Mobile Application Based on Multi-Dimensional Android Feature

SPECIALIZATION: Software Engineering

POSTGRADUATE: Ai Gong

MENTOR: Professor Zhenyu Chen

Abstract

With the wide-spread use of Android applications in people's daily life, the security and quality assurance of Android applications become more and more important. In order to evaluate the potential risks of Android applications, Java static code metrics are introduced to predict the defects of applications. However, the existing technology ignores the specific features of Android applications, and the prediction results have no guidance for the modification of defect code, which eventually leads to the failure to locate and repair the risk vulnerabilities in the applications.

This paper proposes a risk assessment technology for mobile application based on multi-dimensional Android features, which is used to detect the quality and security risks of applications. Among them, quality risk refers to fault tendency, including safety, memory, performance and other risk issues. We summarize the bad practices in Android applications, define 15 Android code smells, and give some suggestions on code refactoring. In order to automatically detect these code smells, we developed a detection tool named DACS. From the dimension of programming criterion, we extract 15 custom code smells and 15 publicly defined code smells. From the four dimensions of size, complexity, duplicate and violation, we extract 21 Java static code metrics. After preprocessing these multi-dimensional features, we combine nine popular machine learning algorithms to build the security risk level prediction model of the application, which helps developers identify high-risk applications. In addition, we select five kinds of code smells and the number of code lines as independent variables, and the number of faults as response variables. Based on the two discrete regression algorithms, we built the fault counting model to study the impact of code smells on the quality of application.

In order to verify the effectiveness of DACS tool, we run the tool on 20 open-source Android applications and gets a detection result. At the same time, we detect these applications manually, and analyze the consistency between the results and the results of the tool. We found that the detection results of DACS are consistent with those of manual detection. Therefore, DACS tool can replace manual detection. In order to evaluate the prediction model, we collected 4575 Android applications from GitHub and conducted three experiments. The results show that (1) Android code smells can improve the risk prediction performance of the model. (2) The prediction model built by random forest algorithm has the best performance (AUC = 0.97). (3) Android code smells such as MIM and LIC may easily cause high risk of applications, so developers should pay attention to them. In addition, we also construct the fault counting model on 645 Android applications. Compared with the negative binomial regression model, the counting model constructed by zero inflation negative binomial regression algorithm is better (AIC = 517.32, BIC = 522.12). Besides, some code smells such as code lines (ncloc), malicious compression (MU), weak encryption algorithm and code lines （WCA:NCLOC ） can significantly lead to the quality risk of applications.

Keywords: Android code smells， Java static code metrics， Android risk， faults

目录

表 目 录

插图

第一章　绪论

1.1　研究背景及意义

在过去的十年里，移动应用在人们的日常生活中扮演者越来越重要的角色。其中，安卓应用在移动应用市场中发展最为迅速。虽然安卓应用的发展给大家带来了便捷的生活方式，然而其快速迭代的开发模式、开发环境的差异以及不完备的软件测试等因素都威胁着移动应用的质量。开发人员在软件开发过程中应该养成规范的代码风格，以提高软件产品的质量。Kitchenham 等人 [1] 将软件质量抽象定义为“需求的满足度”，这个定义清楚地表明，一个高质量的软件需要满足不同维度的需求，包括功能性需求以及非功能性需求。在系统工程和需求工程中，功能性需求描述了特定的行为，可以表示为输出如何响应输入 [2]，它通常会在业务需求说明书中被明确声明。与功能性需求不同，非功能性需求定义了面向对象的系统属性，如可访问性 [3]、可靠性 [4]、可扩展性 [5] 和安全性 [6] 等，它们是软件品质的有力保障，但是却不受开发人员的重视。非功能性需求与软件开发的设计流程息息相关。Fred 在《人月神话》中指出，软件架构在一定程度上决定了软件质量。为了满足非功能需求，面向对象的软件系统应该遵循一套软件设计原则，如抽象 [7]、层次结构 [8] 和封装 [9]。

在实际开发过程中，开发人员往往只关注于功能性的需求，依赖用户所反馈的具体问题进行针对性的修复，而对一些无法具象的非功能性需求置之不理。这些问题虽然不会影响应用程序的正常使用，但会给应用程序的安全和质量带来不利影响。一旦一个移动应用出现安全性的问题，可能会使大量用户的生活受到影响。据我们所知，移动应用程序存在潜在的安全问题并非罕见。正如奇虎的一份报告所述[1]（中国最大的互联网安全公司），18000 个 Android 主流应用程序中 99.5% 面临安全问题，平均每个应用程序有 38.6 个安全问题。此外[2]，只有 4% 的智能手机和平板电脑使用反恶意软件和防病毒软件。因此及时识别和解决移动应用的安全问题具有重要意义。除了安全和隐私性问题外，应用程序的内存、性能、用户体验等各项风险问题也需要我们的关注和研究。在应用程序正式发布前，它们都要经过反复的调试和测试。由于测试的不充分和不完善，导致很多通过测试的应用也会存在各种问题和故障。因此我们需要完成应用程序的质量风险评估，了解哪些因素会增加应用的故障倾向性，并及时修正这些因素。

[1] https://research.360.cn/2015/reportlist.html?list=1

[2] http://www.theregister.co.uk/2011/08/04/

1.2　相关研究现状

1.2.1　安卓应用的代码异味研究

在面向对象设计中识别不良设计和不良编程实践的一种可能方法是检测“代码异味”。Fowler[10] 引入了“代码异味”的隐喻来描述代码中的特定结构，这表明重构是可以被应用的。他们非正式的定义了 22 种代码异味，这些异味可能会在软件系统的进一步开发中导致难以维护的问题。到目前为止，有许多软件分析工具可用于检测代码异味 [11–13]，这些工具通常用于检测代码异味，以分析软件的质量。

在公开发表的论文和期刊中，许多文献都记录了代码异味对软件应用程序的影响。Olbrich 等人 [14] 研究了两种代码异味（万能类和散弹式修改）的演化及其对软件质量的影响。一年后，Olbrich 等人 [15] 研究了另外两种代码异味（万能类和异曲同工的类）和软件缺陷之间的关系。为了量化这些影响的大小，Sjoberg 等人 [16] 使用了回归分析算法来解释 12 种代码异味与维护工作之间的关系。此外，Zhang 等人 [17] 研究了 5 种代码异味（数据泥团、语句转换、夸夸奇谈的未来行、过度耦合的消息链和无用的中间人）对软件故障的影响，并测量了这些影响的大小。

Android 系统的上层应用大多是沿用 Java 语言进行开发的，因此之前的研究都基于 Java 静态代码指标来评估 Android 应用程序的风险。Mario 等人 [18] 使用 DECOR 工具来检测移动应用程序中的几种面向对象的代码异味。他们研究了这些面向对象的代码异味对软件质量指标的影响，特别是与错误倾向相关的指标。然而，传统软件程序和 Android 应用程序中的代码异味的分布是不同的 [19, 20]。在 Android 应用程序中，有许多特定的代码异味比面向对象的代码异味更频繁 [21]。与传统的 Java 程序不同，Android 应用程序有一些自身的特性。它们使用了不同的库和资源，是在不同的设计模式下编程开发的。例如，Android 上的 gui 是通过 XML 声明的，Android 应用程序没有特定的主方法（Android 入口点由事件处理程序处理）。此外，许多 api 是专门为实现一些移动功能（包括联系人、电源管理、图形界面等）而设计的。

关于这些 Android 应用程序的特性，Reimann 等人 [22] 在文章中总结了一组糟糕的编程习惯（例如，一个非静态的内部类蕴含对外部类的引用），即 Android 特定的异代码味。这些 Android 代码异味可能会威胁到移动应用的安全性、数据完整性和软件质量 [22, 23]。Ghafari 等人 [24] 定义了 28 种安全代码异味，这些异味指出了应用程序的隐私和安全性问题。同时，他们基于静态代码分析技术开发了一个轻量级工具，可以检测出其中 10 种安全代码异味。为了缩减应用程

序的能量消耗，Gottschalk 等人 [25] 专注于检测与性能相关的代码异味并重构它们。除了这 28 种代码异味之外，Reimann 等人定义了一组 30 个 Android 特定的代码异味并得到了更广泛的认可。Reimann 等人 [22] 根据异味影响的类型和它们所属的环境对这些异味进行了分类。许多检测工具 [26–28] 也被开发出来用于检测 Reimann 等人定义的一些 Android 代码异味。

Android 代码异味可能威胁到移动应用程序的几个非功能属性，例如安全性、性能和内存。Hecht 等人 [29] 检测到 3 个面向对象的代码异味（依赖性强的类、过长的方法和复杂的类）和 4 个 Android 代码异味（非静态内部方法、内部类泄露、界面渲染过度和不闭合的类）来评估 Android 应用程序的质量。此外，他们还 [23] 研究了修复 3 个 Android 代码异味（Getter/Setter 访问内部字段、非静态内部方法和 HashMap 用法）后对软件性能的影响。Palomba 等人 [30] 研究了 9 种 Android 代码异味对移动应用程序性能的影响。受这些研究的启发，我们决定引入安卓代码异味来评估应用程序的风险，包括安全风险等级的预测和应用程序的质量分析。

1.2.2　安卓应用的风险评估技术

Koenig 等人 [31] 介绍了移动应用程序的漏洞。全面分析了 Android 应用程序及其设备。此外，他们还重点研究了用户如何防范恶意攻击，并基于 Android 系统的设计架构，提出了一些防范措施。利用 ded 反编译器，Enck 等人 [32] 对一千多个 Android APK 进行反向编译，并得到了它们的源代码。通过静态代码分析技术，他们发现了现有的应用程序中存在权限滥用、隐私泄露等危险行为。Chin[33] 等人检查了 Android 应用程序的交互通信并识别出应用程序中的安全风险。此外，他们还开发了一个工具，ComDroid，用于自动化检测与通信相关的安全漏洞。Rahman 等人 [34] 收集了 1407 个 Android 应用程序中的 21 个 Java 静态代码指标，并使用提取到的指标来预测 Android 应用程序的隐私和安全性风险。Qin[35] 等人根据故障触发条件对 bug 进行了分类，为开发人员的代码修复提供指导。Cotroneo 等人 [36] 提出了一套故障建模指南，研究了故障注入对 Android 移动用户体验质量的影响。Mirzaei[37] 提出了一种半自动化的方法，用于 Android 应用程序的异常定位，找出与故障相关的代码片段。

研究者多致力于分析应用程序的质量，解决与权限相关的安全问题 [38]。在本文中，我们的工作是基于多维安卓特征评估应用程序的风险，帮助开发者更好地了解应用程序的质量和安全风险。在应用安全方面，我们研究了应用程序的隐私和安全性问题。在应用质量方面，我们研究了应用程序的安全，内存，性能等各项问题，找出与故障倾向性有关的代码结构，并对它们及时修复和重构。

1.3　本文主要工作

本文基于多维安卓特征对移动应用的风险进行了评估。我们首先自定义了15 种安卓代码异味，描述了代码异味并提出相应的修复意见。同时，我们还开发了 DACS 工具来检测这 15 种代码异味。具体来说，该工具根据定义，针对为每一种代码异味，设计出了相应的检测算法。它将 Java 源码转换成抽象语法树后，基于检测算法去搜索语法树，找出包含代码异味的代码结构。为了验证 DACS 工具在检测代码异味方面的有效性，我们收集了 20 个开源应用程序，邀请人工检测出其中的代码异味，并将人工检测结果和工具检测结果进行一致性比较。

除了上述的 15 种代码异味，我们还引入了 15 种正式定义的代码异味 [22]，研究这 30 种安卓代码异味对应用程序的安全性的影响。具体来说，我们建议将代码异味集成到现有的 Java 静态代码指标中，以便更好地构建应用程序的安全风险预测模型。我们首先研究了 Java 代码指标和代码异味之间的相关性。并且发现大多数代码异味是相互独立的，它们与 Java 代码指标有较弱的相关性。然后，我们以代码异味和 Java 代码指标为多维特征指标，并应用九种流行的机器学习方法，构建了应用程序的安全风险等级预测模型。我们收集了来自 Github 的 4575 个安卓应用程序作为实验数据集来评估我们的模型。结果表明，安卓代码异味可以提高模型的风险预测能力，其中，随机森林（RF）[39] 比其他学习方法的预测性能更好，它的 ROC 曲线下方的面积（AUC）等于 0.97。最后，我们研究了每种代码异味在风险等级预测模型中的重要性，并着重介绍了几个安卓代码异味，提醒开发人员在开发应用程序时应该更加注意这些代码异味。

为了探索代码异味对安卓应用程序质量的影响，并定性计算出这些影响大小，我们试图去拟合代码异味和故障倾向性的关系。其中故障涵盖了应用程序中有关安全，内存，性能等各方面的风险问题。据我们所知，目前还没有（i）大规模研究收集安卓应用程序中的故障，（ii）分析代码异味对应用程序故障数的影响，以及（iii）衡量这些影响的程度。本文分析了 GitHub 平台上 4575 个开源的安卓应用程序，根据故障关键字，统计出每个应用程序在一个版本内的故障数。此外，与代码大小相关的指标和代码异味之间存在关联性。因此，本文总共选取了 5 种安卓代码异味，以这些代码异味和 NCLOC 指标（代码行数）为自变量，应用程序的故障数为因变量，并分别应用负二项回归和零膨胀负二项回归算法建立了故障数计数模型。我们的目标是扩展/补充以前的研究，分析安卓应用程序中的故障，并确定代码异味与故障数之间的关系。

本文的主要贡献如下：

1) 本文提出并定义了全新的 15 种与安卓应用程序安全相关的特征指标——

代码异味，并针对性的给出了修复意见。

2) 本文开发了检测 15 种代码异味（自定义）的工具 DACS，通过将工具检测结果和人工检测结果进行一致性比较，证明工具的有效性。
3) 本文结合安卓代码异味和 Java 静态代码指标，并应用不同的学习算法建立了风险预测模型，其中随机森林建立的模型最好，AUC 可达 0.97。
4) 本文研究了代码异味和 Java 静态代码指标在安全风险预测模型中的重要性。研究结果表明了哪些指标会造成应用的高风险，需要注意修复。
5) 本文爬取了 4575 个安卓应用程序的源代码，对它们的代码全面剖析：提取了 30 种代码异味和 15 种 Java 静态代码指标，并收集了故障信息。
6) 本文研究故障数和代码异味之间的关系，应用两种回归算法建立了故障数模型，探索哪些代码异味会显著增加应用程序的故障倾向性。

1.4　本文组织结构

本文的组织架构如下所示：

第一章 绪论。介绍了与安卓相关的背景与研究意义，分析了安卓应用领域的现有成果，特别是与风险相关的研究，说明了本文的重点工作和突出贡献。

第二章 相关技术概述。主要讨论了与软件质量相关的代码指标，包括各类安卓代码异味，给出了目前流行的检测方法和工具，并说明了安卓代码异味对应用程序的影响是方方面面的。为了定量计算影响大小，本文还调查了指标对故障数的影响，建立了故障数计数模型。

第三章 多维安卓特征的预处理。对安卓特征指标进行了一系列的预处理操作，包括标准化、独立化、降维和类不平衡处理。介绍了每一步操作的原理和算法，使得处理后的指标能更好地应用于移动应用风险评估系统。

第四章 基于多维安卓特征的移动应用风险评估技术。是本文的重点章节，介绍了本文技术框架的细节，定义了全新的与安卓特定的代码异味，并对其修复提出了建议。介绍了基于安卓代码异味构建风险预测模型的过程，同时建立了故障数计数模型，研究了代码异味和应用程序的故障倾向性的关系。

第五章 实验设计与分析。以第四章的技术描述为基础，介绍了本文的实验设计与分析部分。本章节通过设定实验目的，利用不同的应用程序数据集分别完成了三个实验，展示并分析结果，从而更好地评估移动应用风险评估技术。

第六章 总结与展望。主要介绍了本文的工作，从不同的角度分析了风险评估技术存在的局限性。此外，本章节还对论文中的每一章节的工作进行了总结，讨论了未来的研究计划和改进方向。

第二章　相关技术概述

2.1　软件缺陷预测

2006 年，Nagappan 等人 [40] 使用复杂的特征指标来预测组件的故障，这表明如果我们能够确定这些组件在出现缺陷时伴随着哪些特征，我们就可以通过提取组件的这些特征来预测它们的缺陷。同样，我们相信，特征指标的复杂组合也可以用来预测 Android 应用程序的安全风险等级。Barrera 等人 [41] 使用 27 个功能级指标来检验软件内部质量和安全漏洞之间的相关性。Rahman 等人 [34] 评估了静态代码指标（如代码行数、函数复杂度和 McCabe 复杂度）如何用于预测 Android 应用程序的安全风险。Alenezi 等人 [42] 研究了静态代码指标对 Android 应用程序的安全性和隐私性的影响。在软件缺陷预测过程中，正确选择、处理和优化数据集中提取的指标是关键。在软件领域，最流行的度量指标是源代码指标 [43, 44]，但是其他类型的指标在缺陷预测上也被认为是有效的，例如设计指标 [45]、变更指标 [46]、挖掘指标 [40] 或过程指标 [47, 48]。

2006 年，Kent Beck 等人定义了一组独立的设计指标，首次提出了代码异味的概念，并在网站上[1]解释了代码异味的含义：代码异味指示了一处可能发生错误的代码片段，并且这种事件是不确定发生的。一个良好的代码结构也可能被认为是一种代码异味，因为它经常被误用，或者存在一种更简单有效的替代语法。代码异味并不意味着语法错误，它只是一个迹象，表明有必要进行更仔细的检查，以防出现代码漏洞。很多研究都利用代码异味作为软件质量保障的指标，然而其有效性还有待评估 [49, 50]。本文的研究动机主要是研究代码异味是否能有效预测安卓应用程序的缺陷。在工业软件开发中，Holschuh 等人研究了代码异味在面向 Java 编程的缺陷预测中的有效性 [51]，Hryszko 等人研究了在面向.NET 的工业软件项目中，代码异味用于缺陷预测的有效性 [44]。在面向 Android 语言开发的应用中，目前没有人研究代码异味在其风险预测过程中的有效性。因此，我们决定引入安卓代码异味来评估应用程序的隐私和安全性。

2.2　安卓代码异味

代码异味和反模式都代表了一种不良的代码设计模式。反模式通常指的是高度不可靠和无效的设计模式 [31]。Brown 等人 [52] 从三个角度提出了反模式：

[1] http://c2.com/cgi/wiki?CodeSmell

开发人员、架构师和管理者。他们提出了克服反模式和改进软件结构以支持后续扩展和长期维护的方法。与反模式不同，代码异味是源代码中的一个符号，它可能会导致更深层的问题 [53]。代码异味的概念是 20 世纪 90 年代末由 Kent Beck 在 WardsWiki 上提出的，在 Beck 和 Fowler 正式定义了 22 种代码异味之后 [10]，代码异味在工业界和学术界得到了广泛的应用和研究。在 2000 年至 2009 年间发表的所有 39 种异味研究中，Zhang 等人 [49] 发现 Fowler 等人定义的 22 种异味比其他异味研究得更多。

很多开发者都使用 Java/C++ 语言去开发 Android 应用程序。因此面向对象的代码异味也可以用来评估 Android 应用程序的质量。然而，面向对象的代码异味也不能完全表示出 Android 应用程序的结构特性，主要有以下四点原因。（1）与传统的应用软件相比，面向对象的代码异味在 Android 应用程序上有着不同的表现。例如，长方法代码异味在移动应用程序中出现的可能性几乎是两倍 [19]，电脑软件中使用的代码异味的重构方法可能不适用于 Android 应用程序 [54]，外部重复代码异味和内部重复代码异味是电脑应用程序的两种主要代码异味，而 Android 应用程序中有更多不同类型的代码异味 [20]。（2）由于面向对象的代码异味主要是在移动应用出现之前定义的，因此它们没有考虑到安卓应用程序的特殊性，并且与 Dex 文件的字节码 [29] 不兼容。（3）Android 应用程序过于依赖第三方库 [55]，因此本质上比传统应用程序复杂。（4）虽然 Android SDK 兼容了大多数的 Java SDK，但是他们之间还存在差异性。如，在 Android 的四大组件（Activity，Service，Content 和 Provider）中，Android 在 AndroidManifest.xml 中配置属性。在界面部分，Android 只延用了 Java.awt.package 中的 Java.awt.font 等。因此 Android 开发存在特定的配置文件、基础语法和代码结构，这部分信息需要额外的特征指标去描述。基于以上几点原因，当我们研究代码异味对 Android 应用程序的不利影响时，需要考虑一些特定的 Android 代码异味。

目前，Android 应用程序被定义了不同类型的代码异味 [22, 56]。Reimann 等人 [22] 定义了 30 种 Android 特定的代码异味。他们列出了这些代码异味对应用程序有哪些影响，比如效率、用户一致性或安全性等。代码异味来自不同的上下文，如数据库、接口或用户界面。在安卓应用程序中，Hecht 等人 [21] 提出 Android 特定的代码异味比面向对象的代码异味更频繁。然而，在已发表的研究报告中，Android 特定的代码异味却并没有都被研究过。Android 特定的三种代码异味：内部 Getter/Setter（IGS）、非静态内部方法（MIM）和 HashMap 用法（HU）已经被多次研究 [21, 23, 57, 58]。内部类泄露（LIC）、界面渲染过度（UO）和重量级广播器（HBR）也被研究过 [21, 29, 58]。然而其他许多 Android 代码异味被研究得很少。在我们的研究中，我们选择了 15 种很少被研究的 Android 特

有的代码异味。这些代码异味将在4.3.2中介绍。

2.2.1　安卓代码异味的检测方法

安卓应用程序沿用了 Java 语言作为开发语言，因此面向对象的代码异味通常也会被用来衡量 Android 应用程序的质量。有很多方法可以检测代码异味和反模式 [59, 60]，这些方法的灵感大多来源于 Fowler 等人的工作 [10] 和 Brown 等人的工作 [52]。他们主要通过从源代码中提取出结构化指标，如基于源代码指标检测代码异味。Marinescu[61] 和 Zhao[62] 利用与良好设计原则的偏差提出了一种代码异味检测方法，并对检测结果进行分析。Moha 等人 [59] 设计了 DECOR 方法来定义和检测代码异味。如基于源代码的行数检测某些代码异味（过长的函数或过大的类），基于 McCabe 圈复杂度检测某些代码异味（复杂的类）。此外，代码异味总是集中出现在某处代码片段中。因此，Wang 等人 [63] 设计了一种多标签的检测方法，找出同一代码结构中的不同代码异味。

除了利用结构信息来检测代码异味外，添加历史信息也可以帮助捕获互补属性信息。Palomba 等人 [64] 提出了一种称为 HIST（通过历史信息来检测代码异味）的方法，通过从版本控制系统中挖掘出变更历史信息，来检测五种不同的代码异味。他们将 HIST 技术应用于基于 Java 编写的五个不同的开源 Android 应用程序。结果表明，仅靠代码分析方法而无法识别的代码异味，却能被 HIST 方法所识别出。Ouni 等人 [65] 从大中型开源系统中提取开发历史信息来自动化重构代码异味，消除代码异味对软件质量的影响。

除了面向对象的通用代码异味，Reimann 等人 [22] 还定义了 30 种 Android 特定的代码异味。在大多数检测方法中，研究人员首先对代码异味的检测规则进行人工定义，然后将其转换为参数关键字和阈值的组合。例如，Hecht 等人 [29] 开发了一个工具 PAPRIKA，它根据一组人工定义的检测规则，检测出四种 Android 特有的代码异味。基于 Reimann 等人 [22] 提供的代码异味的精确定义，Palomba 等人 [28] 开发了一个 aDoctor 工具，使用源代码的抽象语法来检测 15 种 Android 特有的代码异味。人工定义检查规则是一个耗时的过程，因此 Kessentini 等人 [58] 建议使用多目标遗传编程算法（MOGP）自动生成检测 Android 代码异味的检测规则。MOGP 算法可以计算出检测参数及其阈值的最佳组合函数，并以此检测出代码异味。

2.2.2　安卓代码异味的检测工具

大多数安卓应用程序都使用了 Java 语言进行开发，因此 Java 语言的检测工具可以用来检测安卓应用程序的代码异味。有很多工具都实现了自动化检测面

向对象的代码异味，比如 DECOR[2][59, 66], Checkstyle[3], inFusion[4],iPlasma[5], Stench Blossom[12], PMD[6], 和 JDeodoran[7][67]。在这些工具中，除了 PMD，Checkstyle，Stench Blossom，JDeodorant 工具能指示代码异味的位置信息外，其他工具均未能提供异味的位置信息。除了 JDeodorant 工具能提供代码重构选项外，大多数工具都无法对检测到的异味自动执行重构操作。同时，每个工具可以检测到的代码异味的数量都是有限的。在这些检测工具中，除了 iPlasma 和 inFusion 可以检测到十几种代码异味，其他工具只能检测到个位数量的代码异味。此外，上述检测工具都不是针对 Android 应用程序开发的，只有 Lint[8]是一个针对 Android 应用开发的工具，它基于静态分析技术去检测出应用程序中的安全漏洞。

到目前为止，很少有工具能够检测到 Android 特定的代码异味。Reimann 等人 [22] 定义了 30 种正式的 Android 特定的代码异味。基于开源 Eclipse 框架，他们设计了一个名为 Refactory 的非公开的重构工具。Refactory 工具不仅可以检测到 Android 代码异味，还可以通过执行重构操作来修复这些代码异味。Hecht 等人 [26, 27] 为安卓应用程序设计了代码异味检测器 PAPRIKA。该工具可以检测到 8 种代码异味，其中只有 4 种代码异味是安卓特有的（内部 Getter/Setter、非静态内部方法、无内存解析器和内部类泄漏）。此外，作者仅收集了 15 个开源安卓应用程序，对 PAPRIKA 工具的有效性进行了验证。为了提高 PAPRIKA 工具的可信度，Hecht 等人 [29] 更新了 PAPRIKA 工具，以检测来自不同版本的应用程序的代码异味。最近，Palomba 等人设计了一个用于检测安卓代码异味的轻量级工具，称为 aDoctor[9][28]。该工具利用源代码的抽象语法树，从代码中提取结构属性来检测所有包含代码异味的代码实例。Palomba 等人 [28] 在 18 个 Android 应用程序上验证了 aDoctor，并将该工具检测到的代码异味与手动构建的 oracle 项目进行了比较。结果表明，该工具具备高精确率和高召回率。

针对于代码异味的检测，大多数工具（如 Microsoft 代码分析工具）主要分析了公共程序库并统计相关项目信息，例如，是否背离了 Microsoft.NET Framework 文档中的代码编写准则。根据文档，代码分析中大约有 200 条规则，触发 11 种警告[10]。但是编程和设计规则的设计都是主观的。例如，有些开发人员对长方法

[2]http://www.ptidej.net/download

[3]http://checkstyle.sourceforge.net/

[4]http://www.intooitus.com/products/infusion

[5]http://loose.upt.ro/iplasma/

[6]https://pmd.github.io/

[7]http://www.jdeodorant.com/

[8]http://tools.android.com/tips/lint

[9]http://tinyurl.com/hnm2sla

[10]https://msdn.microsoft.com/en-us/library/3z0aeatx.aspx

和大类的“太长太大”有不同的判断。同时，代码异味的定义并不固定。例如，一些开发人员认为数据类型转换是一种不良的代码结构，但是有些开发人员却不这么认为 [12]。由于代码异味的主观性，许多现有的检测工具检测到的代码异味不一致 [17, 68]。当我们研究自动化工具检测到的代码异味时，有必要给出这些异味的具体定义和实际参数，这有利于其他人分析我们的研究。在本文中，我们使用 aDoctor 工具检测了 15 种 Android 特定的代码异味，这些代码异味的定义和检测规则都有完整的描述 [28]（在 4.3.2节中有更详细的讨论）。

2.2.3　安卓代码异味的影响

通俗来讲，代码异味就是软件中的一个设计漏洞，可能会造成对应用程序的安全风险。Android 应用程序中出现代码异味的频率及其影响的严重程度是不同的。例如，开发人员更容易引入长方法的代码异味，而散弹式修改代码异味更容易被重构 [69]。有很多研究讨论了面向对象的代码异味对 Android 应用程序的影响。Verloop 等人 [19] 使用流行的 Java 检测工具（PMD 和 Jdeodorant）来检测 Android 应用程序中的面向对象的代码异味。他们指出，面向对象的代码异味会给应用程序带来质量问题，尤其是那些与错误倾向有关的问题。

在 Android 应用程序中，Android 代码异味对其产生的影响远大于面向对象指标对其产生的影响 [21]。一些 Android 代码异味的出现可能会造成应用程序的性能低下。基于两个开源的 Android 应用程序，Hecht 等人 [23] 研究了 3 种 Android 代码异味对应用程序的 UI 界面和内存性能的影响。在这三种异味中，MIM 异味对 UI 界面的影响最大。修正 MIM 异味后，应用的延迟帧数减少了 12.4%。通过纠正 HMU 异味，垃圾回收调用的频率可以减少 3.6%。Carette 等人 [57] 在 5 个开源 Android 应用程序上中重构了 3 种 Android 代码异味，显著降低了这些应用程序的能耗。其中一个应用程序的能耗降低了 4.83%。Palomba 等人 [30] 评估了 9 种方法级的 Android 代码异味对应用程序的能耗影响。其中有 4 种代码异味对能耗影响最大，方法中存在这些异味时的能耗是其他方法的 87 倍。当他们对这些代码异味进行重构操作后，应用程序的能耗会明显降低。Android 代码异味对应用程序的影响不仅仅体现在性能上。SAĞLAM 等人 [70] 研究了 Android 应用程序中的代码异味与应用程序的用户评级（即声誉）之间的相关性。他们发现，含有 MIM 代码异味的应用程序往往拥有最差的用户评级。

代码大小与代码异味之间存在相关性 [16]。Sjøberg 等人 [16] 引入了异味密度的概念，根据代码大小调整异味数量。Olbrich 等人 [15] 证实了系统大小与万能类数量（某种代码异味）之间存在着很大的正相关。他们观察到，万能类（某种代码异味）和依赖性强的类（某种代码异味）在大系统中所占比例更大。Mannan

等人 [20] 进行了线性回归分析，找出影响 Android 代码异味的因素。结果表明，Android 应用程序中代码异味的总数与行数、文件数量、错误修复提交总数、新功能提交总数、合并计数和核心开发人员数量有关。zhou 等人 [71] 建议，研究面向对象指标的倾向性的时候应该将类大小视为混杂变量。类大小对面向对象指标的变化倾向性有一定的影响 [71]。因此，在研究代码异味对安卓应用程序的影响时，本文考虑了与应用程序的代码大小相关的指标。

2.3　安卓安全漏洞

在 Android 漏洞领域有很多的研究方向，包括授权机制、系统安全、应用程序隐私和安全性。Gibler 等人 [72] 开发了一个名为 AndroidLeaks 的工具来检测应用程序的信息泄漏问题。Chess 等人 [73] 提出了一种方法来揭示源代码中的安全缺陷。Wu 等人 [74] 提供了一个静态分析范式来检测出 Android 流氓软件。Kumar 等人 [75] 构建了一个基于 GIST 特征的模型来找出存在安全威胁的 Android 应用程序。Bose 等人 [76] 训练了支持向量机（SVM）分类器，用于判断应用程序中的操作是否正常。Shabtai 等人 [77] 提出了一种 Android 移动设备检测框架，连续监视移动设备的功能和事件，应用机器学习算法将操作行为分类为正常或异常。在本文中，我们致力于 Android 漏洞领域中的一个研究方向，即应用程序的隐私和安全性问题。我们试图从静态分析中提取有用的特性来预测 Android 应用程序的安全风险等级。Androrisk[78] 与我们的工作最相关。Androrisk[78] 是一个工具，旨在使用模糊逻辑为 Android 应用程序提供风险分数。在 Androrisk 中，更敏感的权限（即访问位置、SMS 消息或支付系统）和更危险的行为（即共享库、使用加密函数、反射 API）被分配了更高的风险值。Androrisk 的一个问题是，它既不能提供准确的风险评分，也不能为开发人员提供有效的建议来降低应用程序的安全风险。Androrisk 通过检测一些敏感权限和危险行为来测评应用程序，会存在一定的假阳性。大多数由 Androrisk 评判为高风险的应用程序，实际上并不是恶意的。在本文中，我们引入了 Android 代码异味来弥补 Androrisk 的不足。Android 代码异味会为应用程序带来一些安全隐患，但是可以通过代码重构来消除。我们的重点是为开发人员提供高准确度的安全风险评估系统，帮助他们识别出处于高风险的应用程序，并针对性的提供代码修改建议，以帮助他们开发出更安全的 Android 应用程序。

2.4　软件故障预测

软件质量是软件工程领域的重要研究方向之一，也是许多前人研究的课题。这些研究通常会构建故障数模型，使软件工程师能够将开发集中在容易出错的

代码上，从而提高软件质量并更好的利用开发资源。故障数模型有两个重要的组成部分，一是拟合函数，二是故障因子。人们通常采用机器学习算法来训练模型，并收集一些代码指标作为参数输入 [79]。其中，代码指标主要分为过程指标（例如，变更和错误数据）和产品（例如，静态代码数据和代码异味）指标以及与开发人员相关的指标。此外，一些研究 [80][81] 将源代码本身的文本作为指标来评估软件的质量。

Hall 等人 [43] 详细分析了 19 项研究中的故障预测模型的性能，结果表明，基于源代码的垃圾邮件过滤技术 [80][81] 的预测性能相对较好。Zhang 等人 [17] 调查了五种代码异味对软件质量的影响，总结出一些与代码故障显性相关的代码异味。虽然源代码指标和代码异味对软件故障的预测均有贡献度，但是仅使用静态代码指标（通常基于复杂性指标）的故障模型的预测性能相对较差。通过将这两种代码指标结合后，预测性能似乎也没有得到改善 [82]。然而，Zhang 等人 [83] 发现只使用 NCLOC（代码行数）指标的故障模型和只使用面向对象指标的故障模型的预测性能几乎一样好，并且比那些只使用源代码指标的模型性能更好。Ostrand 等人 [84] 报告说，NCLOC（代码行数）数据是一种简单却有价值的故障预测指标。Hongyu[83] 报告说，NCLOC（代码行数）是一个度量故障倾向性的早期通用指标。Zhou 等人 [82] 报告说，当使用不同的代码指标构建故障预测模型时，NCLOC（代码行数）指标的表现优于其他代码指标（除了 Chidamber 和 Kemerer 指标）。在其他个别研究中 [85] 显明，NCLOC（代码行数）指标的故障预测能力较差，并且被其他代码指标所超越。不过总体来说，NCLOC（代码行数）在故障预测中通常是有作用的。

此外，一些研究者报告说，以历史数据为指标的故障模型的预测性能良好 [86][87]。D'Ambros 等人 [88] 特别指出，以前的 bug 报告是最好的故障评估因子。同时，更复杂的指标也表现地更好。特别是，Nagappan 等人 [87] 引入了 "代码变化" 指标，这些指标可以提高故障模型的预测准确度。在模型中使用开发者信息时，少数研究报告了相互矛盾的结果。Ostrand 等人 [89] 报告说，添加开发人员信息并不能显著地提高故障模型的预测性能。Bird 等人 [90] 报告说，当开发人员信息被用作社会变量网络中的一个元素时，故障模型的预测性能会更好。很多表现较好的故障预测模型都使用了组合指标。例如，Shivaji 等人 [91] 将基于过程和基于 SCM 的指标与源代码指标一起使用。Bird 等人 [90] 结合了一系列指标，用于故障预测模型的构建。通过整理现有研究资料，我们发现代码异味可以有效评估软件的故障倾向性。此外，代码行数是一个重要的混杂因素，因此在我们的研究中也考虑到了 NCLOC 指标（代码行数）。在本文中，我们评估了安卓应用程序的质量风险。具体来说，我们研究了代码异味和故障倾向性的关系，

其中故障指代了应用程序中有关安全，内存，性能等各项风险问题。

2.5　故障计数模型

本文研究了故障数与 Android 代码异味之间的关系。故障数是一个非负整数，它全面评估了安卓应用程序的质量，包括性能、内存、安全等所有的软件缺陷，并且它是通过计数而不是排序方式来获得的。计数模型一般是广义的线性模型，用来对数据进行拟合，其中响应变量是计数类型。泊松回归模型是一个经典的计数模型，它有一个先决条件是其均值和方差相等。然而，实际数据的方差往往超过预期值。这种现象被称为过度分散。此外，泊松分布要求观测值是独立的。在反复测量的情况下，组内数据通常是相互依赖的，因此必须考虑组内的相关性。Gupta 等人认为 [92]，如果在不满足上述要求的情况下仍然使用泊松分布模型，那么参数估计将会产生较大的偏差。为了允许观测数据的方差超过平均值，Greenwood 等人 [93] 将泊松回归模型推广到负二项回归模型。各种最大化方法，如 Berndt-Hall-Hall-Hausman（BHHH）可以被用来估计负二项回归的参数。负二项分布由相连的复合泊松分布组成，它的泊松均值服从 γ 分布, 其概率分布公式见 (2.1):

$$\begin{aligned} &\Pr(Y=y)=\frac{\Gamma(y+\tau)}{y!\Gamma(\tau)}\left(\frac{\tau}{\lambda+\tau}\right)^{\tau}\left(\frac{\lambda}{\lambda+\tau}\right)^{y} \\ &y=0,1,\cdots;\lambda,\tau>0 \end{aligned} \tag{2.1}$$

其中 $\lambda = E(Y)$, τ 指代过离散度。Y 表示非负整数的计数型因变量。Y 的方差为 $\lambda+\lambda^2/\tau$。另外当 $\tau\to\infty$ 时，变量的方差与均值相等，此时退化为泊松分布。

对于传统的统计数据，我们一般会使用泊松分布或负二项分布去构建计数模型。然而，在某些情况下，“零”事件发生的概率容易被低估，导致这两种数据分布都无法正确拟合数据。1971 年，Johnson 等人 [94] 发现了这种现象，并称之为“零膨胀”。为了解决经济学中的“零通货膨胀”现象，提出了栅栏模型。栅栏模型包括栅栏-泊松回归（PH）和栅栏-负二项回归（NBH），它们假设响应变量中的零和正变量不是来自同一个数据分布。1992 年，DianeLambert 等人 [95] 提出了一种混合分布模型，叫做零膨胀泊松（ZIP）。在此基础上，Greene 等人 [96] 将该思想应用于负二项分布，相继提出了零膨胀负二项回归（ZINB），其中 BHHH 方法用于计算模型参数的标准误差。零膨胀负二项回归模型将数据分为两份，变量大于零的数据服从负二项分布，变量等于零的数据服从离散零分布，公式如 (2.2) 所示。

$$\Pr(Y=y)\begin{cases} p+(1-p)(1+\lambda/\tau)^{-\tau} & y=0 \\ (1-p)\frac{\Gamma(y+\tau)}{y!\Gamma(\tau)}(1+\lambda/\tau^{-\tau}(1+\tau/\lambda)^{-y} & y=1,2,\cdots \end{cases} \tag{2.2}$$

其中，τ 为指代过离散度，均值和方差分别为 $E(Y) = (1-p)\lambda$ 和 $\text{var}(Y) = (1-p)\lambda(1+p\lambda+\lambda/\tau)$。当 $\tau \longrightarrow \infty$ 和 $\text{p} \rightarrow 0$ 时, 分别服从零膨胀泊松和负二项分布。

除了这些计数模型，还有许多其他计数模型。Andreou 等人 [97] 将双随机泊松过程应用于计数数据回归。2013 年，Cameron 等人 [98] 引入了计数模型的贝叶斯分析，并为计数模型的马尔科夫链蒙特卡洛分析提供了一个入口。Rathore 等人 [99] 提出了一种基于线性组合规则和基于非线性组合规则的集成方法来预测计数数据。Sethi 等人 [100] 使用人工神经网络建立了计数模型。与模糊逻辑回归算法相比，神经网络的效果更好。Gao 等人 [101] 对软件故障预测中的 8 种计数模型进行了全面的实证研究。他们发现修正后的计数模型产生的结果与基本计数模型相似。因此，修正后的计数模型不包括在我们的研究中。

2.6　本章小结

本章节内容主要介绍了与软件风险相关的代码指标，包括面向对象的代码异味和安卓特定的代码异味。介绍了目前流行的代码异味检测方法以及检测工具。同时，以真实的应用程序为例，讨论了安卓代码异味指标对安卓应用程序不同层面的影响。在仅使用 Java 静态代码指标评估安卓应用程序的安全风险时，一些安卓开发的特定语言和用法难以被表达，使得这部分的关键信息缺失，造成风险评估的不准确。通过对以往工作的整理，我们发现现有安全风险分析工具如 Androrisk 的误报率较高，本文即首次引入安卓代码异味指标，应用不同的分类学习算法，构造性能更好的安全风险等级预测模型，进一步说明安卓代码异味可以在一定程度上影响应用程序的风险水平。此外，本文还定量计算了代码异味对安卓应用程序质量的影响，调查了代码异味和故障倾向性之间的关系，包括安全，内存，性能等各方面的问题和故障。本文以故障数为因变量，具体代码异味为自变量建立了故障数计数模型，从而警示开发人员在开发过程中要格外规避哪些代码异味。

第三章　多维安卓特征的预处理

在对数据做分析和挖掘时，通常需要我们根据项目特征去提取出更有效的数据指标。在指标集上使用特征选择可以提高数据模型的性能（例如 [90, 91, 102]）。因此，优化的特征指标集的使用，例如，特征选择，特征组合和特征处理是有价值的。在本文中，我们使用静态代码分析技术，从 Android 应用程序的源代码中提取出混合指标，包括 Java 静态代码指标和代码异味。在实际的项目中，原始的特征数据往往是包含噪点的，会有如下几个问题：（1）特征的规格不同，把它们放在一起比较是没有意义的，同时数据分析结果会受数值大的特征所影响。（2）特征之间存在依赖性，造成一定程度上的信息冗余。同时，特征维度过大，增大模型的复杂度。（3）存在缺失值或者空值，需要一些额外的处理手段。因此，特征的预处理操作是非常重要的，它可以更好地提高数据分析模型的性能 [103]。在对原始数据应用数据预处理技术 [104] 后，可以使特征数据干净、无噪声且保持一致。在本文中，我们首先将缺失或者存在空值的数据删除，接着，本章节将针对以上几个问题，提出相应的特征处理手段。

3.1　特征标准化

标准化是数据挖掘中必不可少的特征处理手段之一，它是将特征数据放在一个小的指定范围内进行缩放。在实际的应用中，由于特征值的大小不同，一个值较大的特征可能会压制另一个值较小的特征。这会使某些算法可能忽略掉一些异常却重要的数据点，无法充分学习到小数值特征数据，造成数据的分析结果不准确。而标准化操作可以打破特征数据的单位约束，将差值很大的原始数据固定到一个特定的区间，其目的是平衡这些特征值的规格大小。数据的标准化没有通用的规则，因此具体的标准化规则主要由用户自行决定的 [105]。现阶段有很多数据标准化的方式，其中一种叫做极差标准化（Min-max 标准化），它会将特征数据通过线性变换映射到固定的区间，一般为 0 到 1 或者是-1 到 1。还有一种叫做零均值标准化（Z-score 标准化），它将所有的属性通过期望和均方差缩放到相同度量标准，使得处理后的属性无量纲化，且大小处于特定范围。此外，转换后的数据分布将不会有所改动。假设有 d 维的原始数据 $Y = \{X_1, X_2, \ldots, X_n\}$。数据矩阵是一个 $n \times d$ 的矩阵如下所示：

$$X_1, X_2, \ldots, X_n = \begin{pmatrix} a_{11} & \ldots & a_{1d} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nd} \end{pmatrix} \tag{3.1}$$

给定一组原始数据 Y，Z-score 标准化公式定义为 (3.2)：

$$x_{ij} = Z\left(x_{ij}\right) = \frac{x_{ij} - \bar{x}_j}{\sigma_j} \tag{3.2}$$

其中 $\bar{x}_j$ 和 σ_{j} 为原始数据的第 j 个特征属性的均值和标准差。Z-score 方法提供了一个统一的尺度，使得不同规格的特征之间的比较有意义。每个特征数据计算出相应的 Z-score 值，但是其原始变量的位置和比例信息会丢失 [106]。

3.2　特征独立化

Fenton 等人 [107] 指出，早期预测模型没有充分考虑特征共线性问题，这将增加因变量的可变性，从而降低预测的质量。如果特征向量中的所有特征都不互相依赖，那么贡献度最小的特征则可以简单地从特征向量中删除。通过应用各种贪婪的特征选择方法，可以找到所有贡献度最高的特征集合，构建最简单却最有效的数据分析模型。然而，在实际项目中，许多特征都相互依赖或依赖于底层的未知变量。因此，一个特征表示了多种类型的信息组合。删除这样的一个特征就意味着删除很多的数据信息。PCA（主成分分析法）是一种多变量分析算法，它是解决特征依赖问题的常见方法。在高维度数据集的情况下，PCA 方法也常作为预处理步骤来降低特征维数。PCA 的目标是找到一组特征值最大的特征向量，并将原始数据映射到这组特征向量上，构建维度更低的且相互正交的新数据。它的本质是利用复杂的离散数学理论将大量可能相关的变量转换成少量的独立的变量。这些变量又称为主成分，它们没有实际的特征意义。PCA 进行降维涉及到将一个或多个最小的主成分归零，从而使数据的低维投影保持最大的数据方差。该值越大，意味着数据映射在这些特征向量后保留了更多的信息。反之代表了包含的信息量就越少，为了达到降低特征维数的目的可以将这部分数据删除。假设有 M 个样本 $\left\{X^1, X^2, \ldots, X^M\right\}$，每个样本 i 有 N 维特征 $X^i = \left(x_1^i, x_2^i, \ldots, x_N^i\right)^T$，PCA 的具体操作流程如下。

(1) 对所有特征进行中心化操作。求出每一个特征的平均值，然后针对所有的样本，每一个特征都减去自身的平均值 (x_1 - $\overline{x_1}$, x_2 - $\overline{x_2}$, ...)。经过中心化处理之后，原始特征的值就变成了新的值。

(2) 计算协方差矩阵 C。假设有两维度特征 x_1 和 x_2。则协方差矩阵 C 如 (3.3)

所示，特征 x_1 的方差如 (3.4) 所示。

$$C = \begin{bmatrix} \operatorname{cov}(x_1, x_1) & \operatorname{cov}(x_1, x_2) \\ \operatorname{cov}(x_2, x_1) & \operatorname{cov}(x_2, x_2) \end{bmatrix} \tag{3.3}$$

$$\operatorname{cov}(x_1, x_1) = \frac{\sum_{i=1}^{M} \left(x_1^i - \overline{x_1}\right)\left(x_1^i - \overline{x_1}\right)}{M - 1} \tag{3.4}$$

当样本的特征维度等于 n 时，它的协方差矩阵的边长为 C_n^2。

(3) 求协方差矩阵 C 的特征值 λ 和特征向量 u。

$$Cu = \lambda u \tag{3.5}$$

特征值 λ 会有 N 个，每一个 λ_i 对应一个特征向量 u_i，将特征值 λ 按从大到小的顺序排序，选择最大的前 k 个，$\{(\lambda_1, u_1), (\lambda_2, u_2), \ldots, (\lambda_k, u_k)\}$。

(4) 将原始特征映射到选取的特征向量上，得到降维后的新 K 维特征。投影之后的新特征是 $\left(y_1^i, \quad y_2^i, \ldots, y_k^i\right)^T$。

$$\begin{bmatrix} y_1^i \\ y_2^i \\ \cdot \\ \cdot \\ \cdot \\ y_k^i \end{bmatrix} = \begin{bmatrix} u_1^T \cdot \left(x_1^i, x_2^i, \ldots, x_n^i\right)^T \\ u_2^T \cdot \left(x_1^i, x_2^i, \ldots, x_n^i\right)^T \\ \cdot \\ \cdot \\ u_k^T \cdot \left(x_1^i, x_2^i, \ldots, x_n^i\right) \end{bmatrix} \tag{3.6}$$

至此，PCA 将原始特征按不同权重进行组合，完成了新特征的创建。它将原始 N 维的特征向量降低为 K 维的新特征向量，同时保证了新特征之间保持相互独立。

3.3　类不平衡处理

在很多分类学习算法中，它都要求训练集中每个类别下的样本数都能被平均分配，样本保持无偏差采样。然而，现实世界中的数据分布往往不是这样，其中一个类可能由大量样本实例表示，而另一个类仅由少数样本实例表示。这就是所谓的类不失衡问题。有几个因素可能会影响分类学习模型的性能。其中一个因素与类别不平衡有关，例如，一个类别的训练数据的数量远远多于另一个类别的数据量。在这种情况下，在描述一个罕见但重要的客观现象时（即，与少数类别相关的信息），分类学习模型可能会忽略掉少数类样本信息，将此类样本点当做异常点处理，从而将少数类样本错误的分类到多数类中去。

从不平衡的数据集中去训练模型通常被认为是一项困难的任务。为了更好地理解这个问题，本文绘制了两幅图3.1。在图3.1(a) 中，多数类（-）和少数类（+）之间存在不平衡的关系，数据集呈现出某种程度的阶级重叠。在图3.1(d) 中表示了一种更良好的学习环境，数据比较平均地分布在不同的类别中。如图3.1(a) 所示，少数类样本可能被错误的划分为多数类，这是因为在数据极端失衡的情况下，少数类样本的最近邻概率是多数类样本的概率非常高，因此少数类样本的错误率通常也会很高。

解决类不平衡问题的一种方法是对训练数据集进行随机重采样。对不平衡数据集进行随机重采样的两种主要方法是（1）从多数类中删除示例（简单随机欠采样），以及（2）从少数类中复制示例（简单随机过采样）。过采样方法虽然为类分布的重新平衡提供了一种简单有效的技术，但也带来了分类模型的过拟合问题。例如，通过这种方式，分类模型可以构造表面上准确的分类规则，但实际上只是覆盖了一个复制的样本。与此类似，简单的随机欠采样随机丢弃了某些多数类样本，在为分类模型带来过拟合问题的同时，也可能会删除掉对模型很重要的潜在的有用数据。为了弥补非启发式方法在模型训练中的局限性，本文引入了一些启发式方法。

合成少数类过采样技术（SMOTE）[108] 顾名思义也是一种过采样技术。与简单随机过采样不同的是，它不再是简单复制一些少数类样本，而是在几个相邻的少数类样本之间构造新值，通过线性算法计算出新的少数类样本。具体来说，假设有少数类样本集 X，我们通过抽取每个少数类样本 x，连接任意或所有 k 个最近邻样本来插入合成样本以进行过采样。例如，如果选定的少数类样本数为 T，需要合成的新样本数为 NT，那么我们需要从 k 个最近邻中选择 N 个样本 y_j，并且沿着每个最近邻样本的位置生成一个新样本 p_j。新样本的计算公式如 (3.7) 所示。

$$p_j = x + rand(0,1) * (y_j - x), j = 1,2,...N \tag{3.7}$$

其中，rand (0, 1) 代表一个随机数，其值在 0 到 1 之间。为了在选定样本和最近邻样本之间的线段中选择一个随机点，SMOTE 计算了两个样本之间的差值，将此差值乘以 0 和 1 之间的随机数，然后将其添加到选定样本中。

虽然 SMOTE 方法通过插值法在一定程度上缓解了模型的过拟合（相较于简单过采样方法而言），但是也存在一些局限性。当它对处于界线上的少数类样本合成时，其最近邻样本大概率也处于边界，甚至属于多数类样本。那么合成样本便会落入到多数类样本领域，使得原本清晰的类别界线被破坏。如图3.1所示，多数类样本（-）侵占了原本属于少数类（+）样本的空间，会覆盖掉原本的模型分类界线（反之亦然，少数类样本也会有类似问题）。为了解决这样的问题，我们

引入了 ENN（Wilson’s Edited Nearest Neighbor Rule）算法应用于 SMOTE 处理后的训练集，作为一种异常数据的清理方法。此方法的应用如图3.1所示。首先，在原始数据集 (a) 中的少数类样本空间内，SMOTE 方法利用线性算法插入一些新样本，得到增量后的数据集 (b)。然后对新插入的样本采用最近邻算法，用最近的三个样本进行预测 (c)，如果预测结果错误（和自身标签不一致），则将该样本删除。最后生成一个类簇间数据平衡的扩增数据集 (d)。

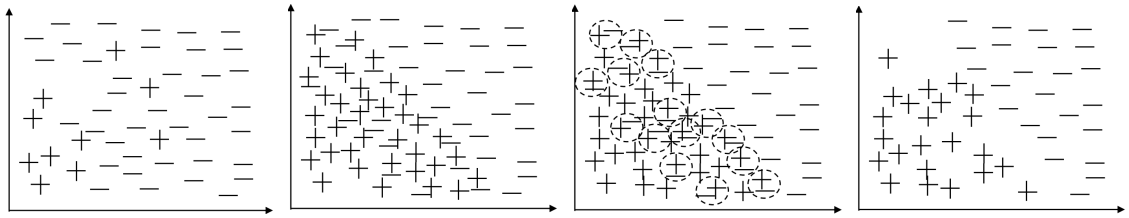


图 3.1: SMOTE + ENN

3.4　本章小节

本章节内容主要介绍了对安卓特征的预处理操作，包括了特征标准化，特征降维和独立化，以及类不平衡的处理。对每一种处理，我们都解释了其原理，并介绍了本文所采用的方法和具体算法。在对特征预处理后，预测模型的安卓特征指标变得尺度规格一致，独立无依赖，平滑无噪，从而能帮助我们更好地构建移动应用风险评估系统。

第四章　基于多维安卓特征的移动应用风险评估技术

4.1　整体概述

这一章将会介绍基于安卓多维特征的移动应用风险评估技术的详细设计。整个设计的框架如图4.1所示，主要有三个研究工作：**(1)** 开发 DACS（Detect Android Code Smells）工具来检测自定义的 15 种安卓代码异味 **(2)** 构建安卓应用程序的安全风险等级预测模型以及 **(3)** 构建应用程序的故障数计数模型。

工作一开发了用于检测代码异味的工具 DACS。本文首先收集了相关文献和资料，自定义了全新的 15 种安卓代码异味，针对每一种代码异味给出了相应的检测规则和代码修复意见。接着基于代码异味定义开发了检测算法，最后将 Java 源代码转换为抽象语法树，并根据检测算法搜索抽象语法树，找出包含异味的代码结构。此外，为了验证 DACS 工具在检测代码异味时的有效性，我们将人工检测结果和工具检测结果进行一致性对比。

工作二评估了应用程序的隐私和安全性风险。本文采用了三步法建立了安全风险等级预测模型。首先，我们从安卓应用程序中提取出 Java 静态代码指标和安卓代码异味。然后我们利用开源评分工具并结合人工审核计算出每个应用程序的安全风险等级。最后，我们将机器学习算法应用于安卓应用程序（以 Java 静态代码指标和代码异味作为特征，以安全风险等级作为分类标签）来构建风险等级预测模型。

工作三评估了应用程序的质量风险，研究了代码异味和故障倾向性的关系，其中故障包含了安全，内存，性能等各项风险问题。本文首先以关键字匹配的方法，统计出每个安卓应用程序的故障数。其次，基于工作二的研究结果，本文选取了其中五种代码异味，以这些代码异味和 NCLOC 指标（代码行数）为自变量，应用程序的故障数为响应变量，应用负二项回归算法和零膨胀负二项回归算法，建立了故障数计数模型，研究了代码异味对应用程序质量的影响及影响程度。

4.2　DACS 工具

本文开发了自己的工具（DACS）来自动检测自定义的 15 种 Android 应用程序源代码中的目标代码异味，原因如下：

（1）这 15 种代码异味是我们自定义的，在现有的异味检测工具中，没有一个工具能够检测出我们正在调查的 15 种异味。

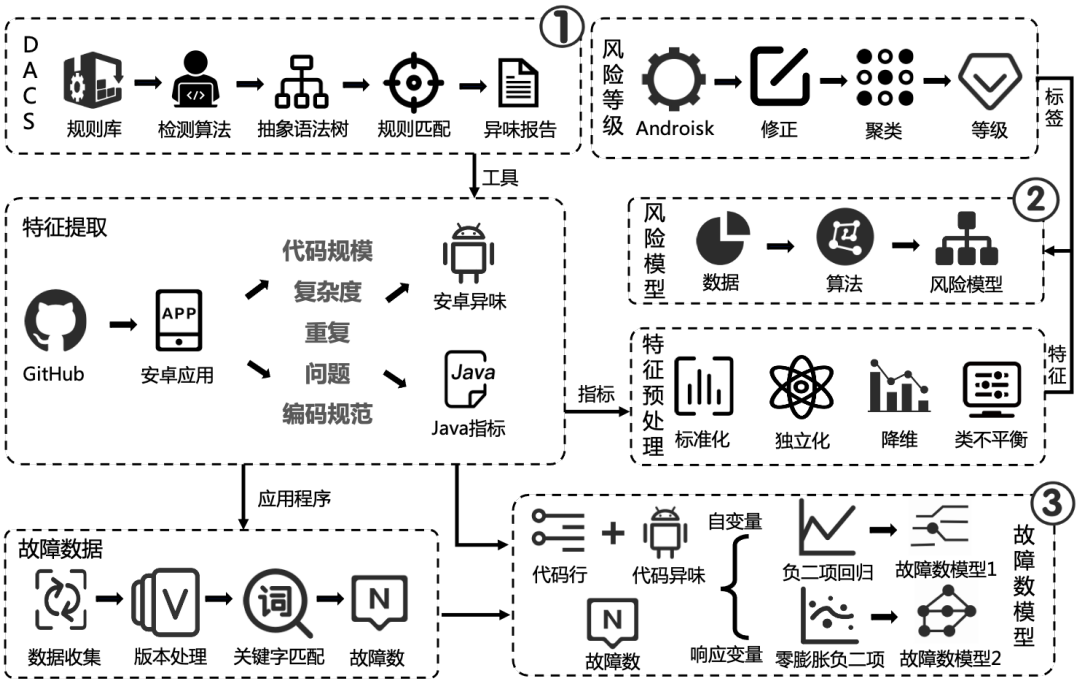


图 4.1: 基于多维安卓特征的移动应用风险评估框架

（2）Fontana 等人 [68] 研究了五种现有代码异味检测工具，发现这些工具检测到的代码异味都不一样。即，现有的工具检测到的异味并不一致。此外，目前的检测工具大多只能判断出包含异味的代码结构，而不能针对异味提出相应的代码重构方案。

（3）代码异味是主观的，目前很少有普遍接受和使用的代码异味定义。不同的工具和研究对同一异味的定义不尽相同。代码异味的定义通常是基于研究人员的个人理解和感受，并且这些定义没有文档记录和描述，因此。很难基于现有开源代码进行修改从而完成适配性工作。很多工具设置阈值判断代码结构是否包含异味，而每个人对阈值的设定带有偏差。例如，函数要多“长”才能算作一个代码异味。这些阈值的不同带来了工具的不一致性。因此，本文决定开发自己的工具来检测代码异味，并将代码异味的定义和阈值记录于文档中，以便于未来的拓展研究。

4.2.1　自定义的安卓代码异味

本文调查了若干与安卓安全相关的技术博客和期刊会议，从编程规范的维度出发，总结并定义了 15 种全新的安卓代码异味，同时我们还针对每一种代码异味给出了相应的修复建议。具体的细节如下所示。

(1.) 弱加密算法 Weak Crypto Algorithm(WCA)

定义：安卓应用程序中经常会出现弱加密的现象，该现象的产生原因主要有两点 [109]：一是使用了弱加密算法；二是使用了强加密算法，但在加密的实现上有漏洞。**1.** 当方法使用弱加密哈希函数（如 *MD2*，*MD4*，*MD5*，*SHA-1* 或 *RIPEMD*）时，会触发代码异味。**2.** 当方法执行 *DES* 算法或在 *AES* 初始化时使用 *ECB* 模式时，则产生代码异味。**3.** 当方法使用密钥长度小于 512 位的 *RSA* 算法，则会检测到异味。**4.** 加密算法建议使用 *Cipher.getInstance(RSA/ECB/OAEPWithSHA256And-MGF1Padding)*，不然黑客会借助已收受的包发送危险请求，产生代码异味。

修复意见：1. 建议开发人员使用 *SHA-256* 和 *SHA-3* 函数，而不是弱加密哈希函数。**2.** 建议采用 *AES* 算法，并指定形式为 *CBC* 或 *CFB*，同时用 *PKCS5Padding* 进行填充。**3.** 建议使用 *RSA* 算法时的密钥长度为 2048 位。

(2.) 不正确的证书验证 Improper Certificate Validation(ICV)

定义：安卓提供了一个内置过程来验证由 CA 签署的证书。当一个证书是自签名的，操作系统会让应用程序完成验证过程。但是，开发人员经常不能正确地实现证书的验证。所以 SSL/TLS 上的通讯通道容易被中间人袭击。**1.** 使用 Webview 组件进行 HTTPs 通讯时，当在 *onReceivedSslError()* 方法中简单地调用 *proceed()* 函数解决证书故障时，会产生代码异味。**2.** 当开发人员继承 X509TrustManager API，重写 *checkClientTrusted()* 和 *checkServerTrusted()* 方法，但在方法体内没有关于审查证书的逻辑代码（空实现）时，会产生代码异味。**3.** 当开发人员继承 HostnameVerifier API，在子类中重写 *verify()* 方法时，没有审查主机名的有效性 (简单输出 return true)，则会检测到异味。**4.** 当应用程序设置不安全的 HostnameVerifier 参数: *SSLSocketFactory.ALLOW_ALL_HOSTNAME_VERIFIER* 时，它会跳过所有的证书验证流程，等同于不做任何验证逻辑，因此代码异味也会被检测出来。

修复意见：1. 建议开发人员不要自定义 *onReceivedSslError()* 方法，也不要对证书问题不作处理，以避免通信数据的泄露。**2.** 对于有权威机构签署证书的 HTTPs 网站，开发者可以使用安卓系统自带的证书验证机制，无需自行实现。**3.** 在 HostnameVerifier 中补充 HTTPs 的域名校验逻辑，即，验证连接到 HTTPs 站点的域名和 SSL 证书中设置的域名是否保持统一。**4.** 建议开发人员延续安卓内置的 HostnameVerifier 参数: *SSLSocketFactory.STRICT_HOSTNAME_VERIFIER)*。

(3.) 无约束的通信 Unconstrained Inter-Component Communication(UICC)

定义：当在 AndroidManifest.xml 中设置属性 *exported* 或 *IntentFilter* 时，应用程序能够重用第三方组件（Activity/ContentProvider/Service/BroadcastReceiver）。**1.** 当组件的 *exported* 被置为 true，亦或未设置该值但设置了 *IntentFilter* 时，该组件有被恶意软件调用的风险，会产生一种代码异味。

修复意见：1. 开发人员应该将 *exported* 的值设置为 false，以便外部程序无法调用组件。如果开发人员希望特定程序能访问外部组件，那么他们需要将 *exported* 属性设为 true，同时定制权限字符串并赋值给属性 *permission*。

(4.) 自定义 Scheme 通道 Custom Scheme Channel(CSC)

定义：AndroidManifest.xml 中的属性 *IntentFilter* 用于解析隐式 intent。*IntentFilter* 有三个参数：*action*、*category* 和 *data*。**1.** 假设 *data android:scheme* 被指定，则启动 intent 类完成组件间的通信，攻击者可以借助网页访问 APP 的组件，产生代码异味。**2.** 假设浏览器能够解析 *Intent URL*，且未设置筛选准则，那么攻击者可能会借助 JS 代码读取浏览器的文件（无论私用或公开），例如窃取 cookies 文件，并产生代码异味。

修复意见：1. 开发人员不应该指定 *data android:scheme* 属性。**2.** 假如函数 *Intent. parseUri()* 被调用，那么 intent 一定要设置严格的过滤条件，同时 intent 必须要满足三种属性：*addCategory(* "*android.intent.category.BROWSABLE*" *), setComponent(null), setSelector(null)*。

(5.) 头文件攻击 Headers Attachment(HA)

定义：在使用 HTTP 请求服务器通信时，开发人员经常向服务器发送一些数据，如平台号、通道号、系统版本号等常见信息。这些数据可能是敏感的，并且依赖于报头传输。**1.** 对用于存储私有数据的头文件的调用被认为是一种代码异味。共有三种类型的头附件：设置 OkHttpClient 的 http 头参数、设置 HttpURLConnection 的 http 头参数以及设置 HttpClient 的 http 头参数。

修复意见：1. 在对第三方服务进行身份验证之前，开发人员不应在报头中存储敏感数据。

(6.) 暴露的剪贴板 Exposed Clipboard(EC)

定义：在安卓应用程序中，系统提供了 ClipboardManager 类用于实现剪切板的功能。它可以暂存数据，将数据保留到内存中。应用通常调用 *settext()/getText()* 或 *setPrimaryClip()/getPrimaryClip()* 函数来将数据更新/读取到剪贴板。**1.** 在剪贴板中存储私有数据（尤其是密码）是不安全的，剪切板中的数据可以被任何程序所读取，因此我们将这种行为定义为代码异味。

修复意见：1. 开发人员应该避免在剪贴板中存储数据。

(7.) WebView 域远程代码执行 Broken WebView’s Sandbox(BWS)

定义：在 Android SDK 中，WebView 组件实现了手机浏览器，用户通过移动应用能够访问网络页面。。其中，WebView 的滥用容易引起远程代码的随意执行。该漏洞的出现原因主要有三个：**1.** WebView 组件中的 *addJavascriptInterface*

接口生成 Java 对象，JavaScript 能够调用该对象的方法，完成和当前 APP 之间的通讯。当 JS 拿到对象后，它可以做任何事情，包括读取设备的 SD 卡上的敏感信息，会产生一种代码异味。 **2.** *searchBoxJavaBridge_* 的 Js 接口会生成 JS 映射对象，从而允许远程程序执行，触发代码异味。 **3.** 同理，*accessibility* 和 *accessibilityTraversal* 的 Js 接口也会引发任意代码的执行，产生代码异味。

修复意见：1. 在 Android4.2 版本之前，通过 *prompt()* 方法处理返回给 JS 的结果；在 Android4.2 版本之后，Google 提供了注解 *@JavascriptInterface*，有效规避了上述风险。**2.** 调用 *removeJavascriptInterface()* 方法删除 *searchBoxJavaBridge_* 接口。**3.** 同理，删除 *accessibility* 和 *accessibilityTraversal* 接口。

(8.) Webview 域明文保存密码 WebView Plain Secret(WPS)

定义：在 Android 操作系统中，WebView 组件默认保存密码：*WebView.setSave-Password(true)*。**1.** 当用户填写密码，且同意保存密码时，密码将以明文形式存储在数据库中，并且会产生代码异味。

修复意见：1. 建议用户关闭密码保存功能：*WebView.setSavePassword(false)*。

(9.) WebView 域约束不严格 WebView Domain Not Strict(WDNS)

定义：安卓内置 *setAllowFileAccess()* 方法来设置 WebView 组件能否采用文件协议，默认设置为 true。在使用文件协议的前提下，一旦以下属性被置为 true 时，代码异味就会被检测出来。**1.** *setAllowFileAccessFromFileURLs()* 方法用于设置是否使用 file 路径中的 JS 代码去读取本地文档。**2.** *setAllowUniversalAccess-FromFileURLs()* 方法用于设置是否使用 file 路径中的 JS 代码去访问跨域源（如 http 域）。**3.** *setJavaScriptEnabled()* 方法用于设置是否允许加载 JavaScript。

修复意见：对于一般的应用程序，开发人员应禁用文件协议，即 *setAllow-FileAccess(false)*；针对必须要借助文件协议实现特定需求的应用程序，开发人员应将上述属性设置为禁止状态。

(10.) 数据备份 Data Back Up Any(DBA)

定义：在 Android 配置文件中，*allowBackup* 属性用于处理 Android 应用程序的数据存档。**1.** 当该值等于 true 时，ADB 调试工具可以绕过超级管理员认证而对数据进行拷贝和导出操作，攻击者能够使用 USB 调试获取敏感数据，从而导致用户隐私泄露，我们定义该行为会产生代码异味。

修复意见：1. 开发人员应该将 AndroidManifest.xml 文件中的 *allowBackup* 属性设置为 false。

(11.) 全局文件可读/可写 Global File Readable/Writeable(GFRW)

定义：在安卓应用程序中，Activity 组件调用 *openFileOutput(String name,*

int mode) 函数将数据导出到文件中。两个参数分别代表了文件的名称，以及文件的操作模式。 **1.** 当操作模式赋值为 *MODE_WORLD_READABLE* 时，攻击者能够读取文件中的敏感信息，这种行为会产生异味。 **2.** 当操作模式赋值为 *MODE_WORLD_WRITEABLE* 时，攻击者能够任意修改文件内容，破坏数据的正确性和有效性，这种行为会产生代码异味。

修复意见：开发人员应确认敏感数据有无存储在文件中。如果存在敏感数据，则应该 **1.** 禁止将模式设置为 *MODE_WORLD_READABLE* **2.** 禁止将模式设置为 *MODE_WORLD_WRITEABLE*。

(12.) 配置文件可读/可写 Configuration File Readable/Writable(CRW)

定义：*SharedPreferences(String name, int mode)* 是安卓系统中常用的数据存储方法。这个方法要求输入两个参数，分别表示文件的名称，以及文件的操作模式。**1.** 当文件模式为 *MODE_WORLD_READABLE* 时，第三方应用程序可以读取文件，从而造成私有数据的泄露，产生代码异味。 **2.** 当文件模式为 *MODE_WORLD_WRITEABLE* 时，第三方应用程序可以删改文件，有可能会引入恶意代码，产生代码异味。

修复意见：开发人员应该指定 *getSharedPreferences()* 方法中的文件模式为 *MODE_PRIVATE*。

(13.) 恶意解压缩 Malicious Unzip(MU)

定义：在安卓程序中解压缩 zip 文件时，将采用 ZipInputStream 和 ZipEntry 类。ZipEntry 类中提供了一个方法 *getName()* 用于读取压缩文件的名称。**1.** 当开发人员解压文件时，他应该对文件名进行检查，确定文件名是否有像../这样的特殊字符串，否则会检测到一种代码异味。

修复意见：1. 解压文件时，开发人员需要过滤上层目录字符串，对文件名做检查。

(14.) SD 卡访问 SD Visit(SDV)

定义：在安卓应用程序中，程序调用 *getExternalStorageDirectory()* 方法将公共数据保存在共享的存储器中。 **1.** 当开发人员将敏感信息存储在外部 SD 卡上且不加密时，将会被检测到代码异味。

修复意见：开发人员应该禁用 *getExternalStorageDirectory()* 方法，将私有数据保存在本地路径下，同时对私有数据加密。

(15.) 不安全的随机数 Unsecure Random(UR)

定义：SecureRandom 类用于获取安卓加密算法中的随机数。它默认应用的是 dev/urandom 生成随机数列，且生成结果是难以推算的。但是当设置种子

时，随机数的生成基于了固定的算法和种子，因此很容易被预测。 **1.** 在调用 SecureRandom 类时，用以下方法生成随机种子会产生代码异味：*SecureRandom.SecureRandom(byte[] seed)*、*SecureRandom.setSeed(byte[] seed)* 和 *SecureRandom.setSeed(long seed)*。

修复意见：1. 建议开发人员禁用 Random 类生成随机数。 **2.** 在调用 *SecureRandom* 时不设置种子。

4.2.2　DACS 工具的设计与实现

我们的 DACS 工具是基于 Android 应用程序源代码的抽象语法树，并根据代码异味的精确定义，匹配检测规则来检测异味的。我们首先利用开源工具 Eclipse JDT (Java Development Toolkit) APIs[1]将 Java 源代码转换为抽象语法树。然后，基于目标代码异味的定义，实现了检测规则算法来搜索语法树。图4.2是代码异味检测工具——DACS 的架构图。

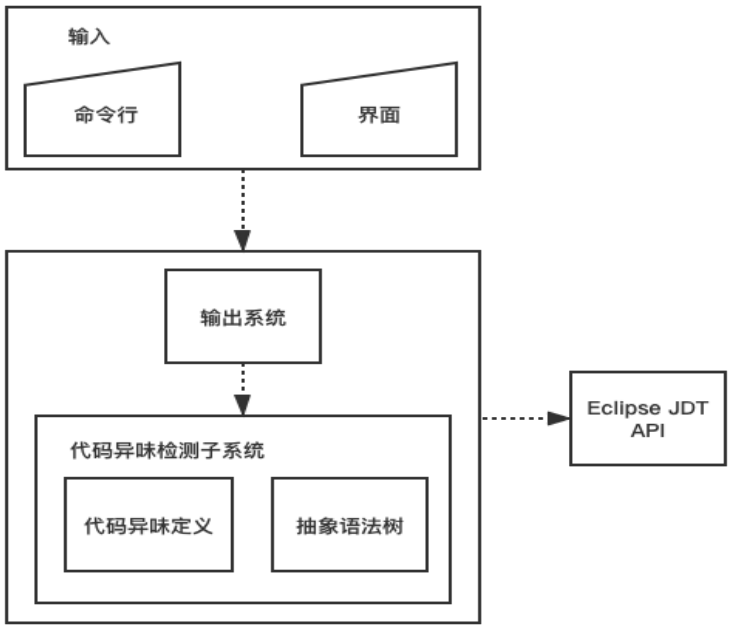


图 4.2: DACS 工具架构图

该工具有命令行和图形化界面两种输入方式，且有独立的代码异味检测系统和输出系统。根据4.2.1节对 Android 代码异味的定义，DACS 工具针对每一种异味实现了检测子系统，并实现输出系统以输出目标异味的检测结果。其中，每

[1]http://www.eclipse.org/jdt/

类异味的检测规则都作为 Android 异味检测子系统的独立类实现，且输出系统将结果（代码异味的数量）以 csv 文件或界面化的形式展现出来。DACS 系统均依赖于开源工具 Eclipse JDT API，以便（i）将所分析的应用程序中的源代码转换为抽象语法树，以及（ii）搜索抽象语法树并计匹配异味检测规则，判断哪些代码存在代码异味。

具体来说，DACS 工具的实现包括如下三个步骤：

步骤一：定义代码异味

详见4.2.1节。

步骤二：将 Java 源代码转换为抽象语法树

为了方便检测 Android 源代码中的代码异味，我们需要先将源代码进行结构化处理。抽象语法树（AST）是一颗 n 叉树，它以树状图的形式对源代码进行了抽象化处理，树节点代表了某种代码语言的组织构造，以更高层次的抽象形式提炼出语法信息，特别是代码中的括号，冒号等不具备有效信息的符号不会以节点的形式表现出来，这也从一定程度上去除了冗余信息，加速了搜索抽象树的过程。

Eclipse JDT 是以 Eclipse 为开发环境的工具插件，它提供了一些开源接口，用于支持 Java 应用程序的开发。Eclipse AST 是解析 Java 代码的核心组件，它被用于构建 AST 树状结构，并提供了一系列类用于解析、构建和遍历 AST 节点。在本文中，我们使用了开源的 Eclipse JDT APIs（版本 3.7.1）将应用程序中的 Java 代码抽象成语法树。该工具开源，功能强大稳定，且使用简单。通过在项目中引入第三方依赖，重写 AST 方法，即可实现抽象语法树的构建。在一个 Java 文件中，主要包含了三块代码结构：包申明 (package{1})、包引用 (import{*}) 和类申明 (class{*})。在类中，也包含三块组件：成员变量 (field{*})、成员方法 (method{*}) 和内部类 (innerclass{*})。在一个 Java 文件中，如果类没有被关键字所修饰，那么一个 Java 文件中允许存在多个类，否则只有一个类。如图4.3所示，展示了 AST 节点的结构图，解释了用 Eclipse JDT 工具将 Java 语言解析成语法树的进程。

步骤三：在抽象语法树中检测代码异味

基于步骤一中的定义，我们为每个代码异味开发了 Java 检测算法。这些算法通过遍历抽象语法树，搜索包含以上 15 种异味的代码结构。举例来说，清单 4.1 展示了针对弱加密算法（WCA）代码异味的 Java 检测算法。算法的输入是一个 Java 类，输出是类中 WCA 代码异味的数量。算法的流程大致如下：（1）根据第4.2.1节的定义，总结出检测 WCA 异味的关键字。（2）以方法级为检测粒度，通过匹配每个方法体内的关键字，判断方法体中是否包含代码异味。（3）计算出

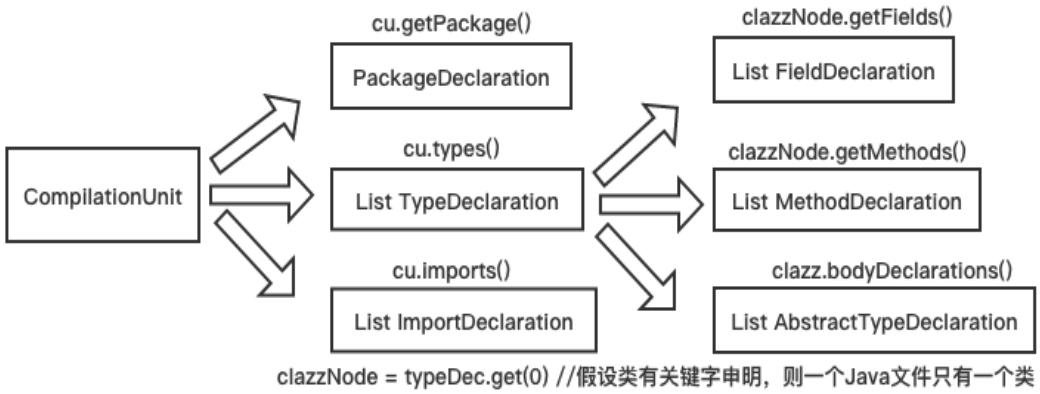


图 4.3: AST 节点的结构图

类中存在的 WCA 异味总数。对于每个.java 文件，我们都进行了代码异味检测，并将异味个数进行累加，最终输出每个 class 文件中每一种代码异味的数量。

```
public int isWeakCryptoAlgorithm(ClassBean pClassBean) {
    int cnt = 0;   //WCA代码异味的数量
    Pattern pattern = Pattern.compile("initialize");
    for (MethodBean method : pClassBean.getMethods()) {
        String content = method.getTextContent();
        if (content.contains("MD2") || content.contains("MD4") ||
            content.contains("MD5") || content.contains("SHA-1") ||
            content.contains("RIPEMD"))
                cnt++;
        if (content.contains("DES") || content.contains("AES/ECB"))
            cnt++;
        if(content.contains("RSA")){
            if(!content.contains("RSA/ECB/OAEPWithSHA256AndMGF1Padding"))
                cnt++;
            Matcher matcher = pattern.matcher(content);
            while (matcher.find()) {
                if (Integer.parseInt(matcher.group(2)) < 512)
                    cnt++;
            }
        }
    }
    return cnt;
}
```

清单 4.1: 弱加密代码异味的检测算法

4.2.3　DACS 工具的使用

工具有两种启动形式：命令行和图形用户界面。我们将 DACS 项目打包成 jar 文件，且打包时没有指定程序入口，因而需要用 java -cp 命令来指定主类。该工具可通过以下命令通过命令行执行：

DACS 启动脚本

```
java -cp DACS-1.0-SNAPSHOT.jar RunAndroidSmellDetection <project-path> <output-path> <code-smells>
```

其中，RunAndroidSmellDetection 是 DACS 项目的主类，<project-path> 是一个字符串，指代需要检测的 Android 应用程序源代码的文件夹，<output-path> 是一个字符串，表示打印代码异味的检测结果的路径，<code-smells> 指代需要检测的代码异味的字符串。这种命令行的运行方式可以方便外部接口的调用，适用于数据挖掘和分析工作。

我们还提供了一个图形用户界面如图4.4所示。DACS 工具允许用户设置运行分析时所需的参数，包括项目源码所在的文件夹和保存运行结果的 csv 文件。此外，用户可以通过勾选框，选择他想要检测的代码异味。当用户点击“开始检测”按钮后，工具立刻开始检测工作。DACS 工具对应用程序中的每个 class 文件进行代码异味检测，并在完成检测任务后，将每一种代码异味的数量输出。当用户点击“输出结果”按钮后，如图4.5所示，代码异味的检测结果可以以图形化的形式展示，输出每一个 class 文件中的每种代码异味的数量。由于检测结果的数据量较大，对目标信息的查询变得更加困难。如图4.6所示，DACS 工具还提供了按类名过滤代码异味的功能，帮助用户快速检索到某个 class 文件的检测结果。

对于 DACS 的输出系统，它负责调用 Android 异味检测子系统的接口，用于输出每种目标代码异味的检测结果。具体来说，检测结果汇总在一个 csv 文件（详见图4.7），其中：

1) csv 文件的每一行表示所分析的应用程序的某种代码异味的数量；
2) 每行的第一列指定代码元素的粒度（即类或方法）；
3) 每行中从 #2 到 #n 的列报告一个整数值，指示在某个代码粒度下，存在的代码异味数量的累积量（例如，如果某个 class 文件存在多个弱加密算法异味时，则将该异味数量累加）。

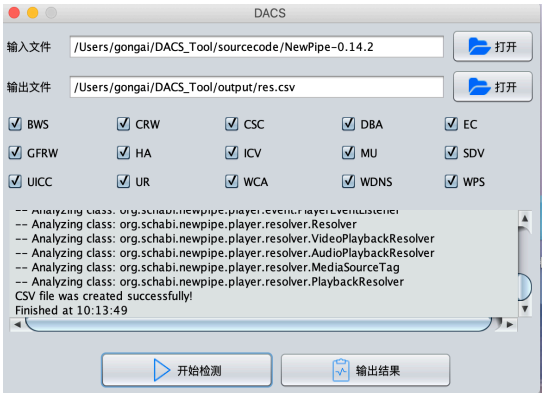


图 4.4: DACS: 配置视图

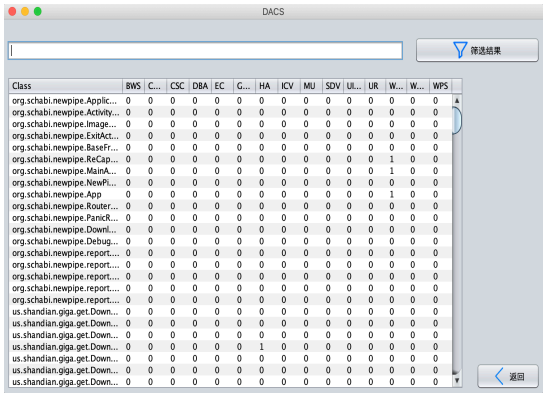


图 4.5: DACS: 输出视图

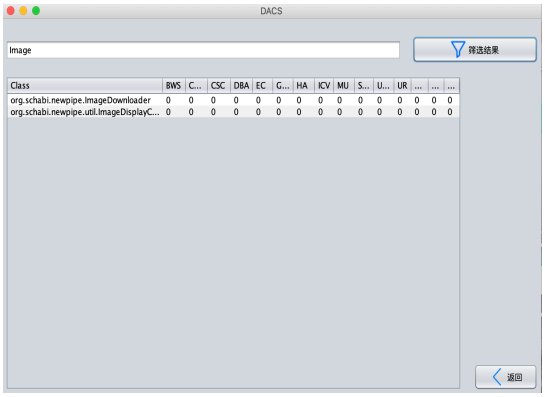


图 4.6: DACS: 按类名过滤代码异味

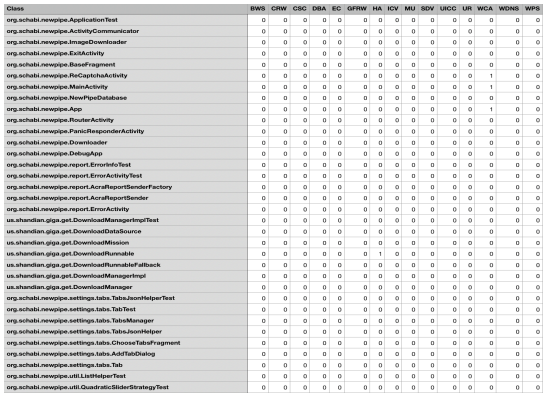


图 4.7: DACS: 输出.csv 文件

4.3　安全风险等级预测模型

本章节将详细介绍本系统关于工作二的内容，即构建应用程序的安全风险等级预测模型，评估安卓应用程序的隐私和安全性。模型的构建包含了几个主要步骤：特征指标和标签的获取，特征的预处理和预测模型的构建。

4.3.1　原始数据的收集

我们的实验项目是从 GitHub 上抓取的[2]。具体来说，我们首先从 GitHub 通过关键字“Android application”搜索项目列表作为候选项目。然后从这些候选项目中，我们将选取用 Java 语言开发的应用程序作为我们的实验项目。实验总共收集了 4575 个开源的安卓应用程序，并对这些应用程序的安全风险程度进行了检测和评估。这些开源应用均是 GitHub 平台上星级较高的应用程序，来自不

[2]http://github.com

同的领域，规模也不同。且它们的类型丰富多样，其类型分布如图4.8所示。这4575 个应用共涉及学习、工具、娱乐、读书、社交、相机和地图等多种类型，覆盖了应用市场上绝大多数的应用类型。

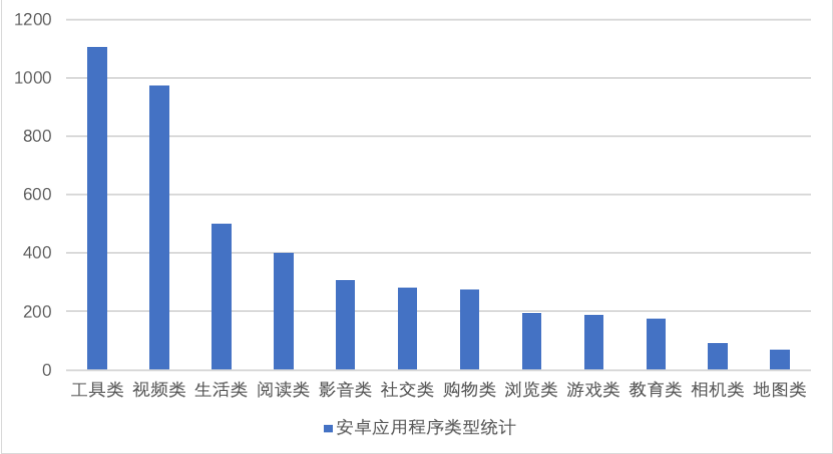


图 4.8: 应用程序类型统计

4.3.2　代码指标的提取

本文总共研究了 51 种安卓代码指标，其中包括了 Sonarqube 定义的 21 种 Java 静态代码指标，Reimann 等人 [22] 定义的 15 种安卓代码异味，以及自定义的 15 种安卓代码异味（详见4.2.1）。本文总共从五个维度去收集安卓特征，其中 Java 静态代码指标可以被细分为四个维度，分别是代码规模，代码复杂度，重复代码和问题。安卓代码异味则是以编程规范为度量维度。

(1) JAVA 静态代码指标

现有的研究发现，一些 Java 静态代码指标，如统计指标（类数、方法数、代码行数等）、复杂度指标（函数复杂度、类复杂度、文件复杂度等）和 OOP（面向对象编程）属性在预测 Android 应用程序的安全风险等级方面有很大的作用。在本文中，我们沿用了先前研究中所使用的指标来帮助预测 Android 应用程序的安全风险等级 [34]。在我们的研究中总共使用了 21 个 Java 静态代码指标（详细信息见表 4.1)。对于每一个 Android 应用程序，我们使用开源 SonarQube 的工具[3] 从不同度量维度检索出这 21 种 Java 静态代码指标（SonarQube 是一个用于连续分析和评估代码质量的开源工具。它可以检测重复的代码、潜在的 bug、代码样式问题以及项目的其他问题)。

[3]http://www.sonarqube.org

表 4.1: Java 静态代码指标

度量维度	指标	描述
代码规模	# 行数 (lines)	代码的行数
	# 函数 (functions)	函数的数量
	# 类 (classes)	类的数量
	# 文件 (files)	Java 文件的数量
	# 文件夹 (directories)	有项目的文件夹的数量
	# 注释行 (comment_lines)	包含注释的行数
	# 非注释行 (ncloc)	非注释代码行数
	注释行密度 (comment_lines_density)	注释行的比率
复杂度	函数复杂度 (function_complexity)	所有函数的平均复杂度
	类复杂度 (class_complexity)	所有类的平均复杂度
	文件复杂度 (file_complexity)	所有文件的平均复杂度
	复杂度 (complexity)	圈复杂度
问题	# 阻断违规 (blocker_violations)	阻断违规数
	# 严重违规 (critical_violations)	严重违规数
	# 主要违规 (major_violations)	主要违规数
	# 次要违规 (minor_violations)	次要违规数
	# 违规 (violations)	违规数
重复代码	# 重复块 (duplicated_blocks)	重复行块数
	# 重复文件 (duplicated_files)	重复的文件数
	# 重复行 (duplicated_lines)	重复的代码行数
	重复行密度 (duplicated_lines_density)	重复代码行的比率

(2) 安卓代码异味

如2.2部分所述，Android 代码异味可能威胁 Android 应用程序的非功能属性，并有助于提高安全风险预测模型的准确度。以编程规范维度出发，我们引入了安卓代码异味来评估应用程序的代码结构是否存在漏洞。在本文中，我们研究了 15 种 Android 代码异味用于（详见表 4.2）预测 Android 应用程序的安全风险等级。这些 Android 特有的代码异味是由 Reimann 等人 [22] 所公开定义的流行的代码指标。本文通过 aDoctor[28] 工具来检索这 15 种代码异味。aDoctor 一个轻量级代码异味检测工具，平均精度为 98%，平均召回率为 98%，同时 aDoctor 工具对于这些代码异味的定义和检测标准在其论文中 [28] 也有详细说明。

表 4.2: 安卓代码异味

指标	描述
DTWC	无压缩的数据传输
DR	一个应用程序被设置为可以在 Android 中进行调试
DW	未正确释放 Wakelock 锁机制
IDFP	低效的数据格式和解析器
IDS	低效的数据结构用于映射整数
ISQLQ	低效 SQL 查询
IGS	使用 Getter 和 Setter 访问内部字段
LIC	非静态嵌套类保存对外部类的引用
LT	一个线程没有被充分释放
MIM	不访问内部属性的方法不能成为静态方法
NLMR	没有用于清理不必要资源的方法
PD	私有数据暴露给其他应用程序
RAM	刚性报警管理器：报警管理器触发的操作唤醒手机
SL	不使用 for 循环的增强版本
UC	Closeable 类不调用 close 方法

除了以上定义的 15 种代码异味，如第4.2.1节所示，本文还自定义了 15 种安卓代码异味。我们使用了自己开发的 DACS 工具来检测这 15 种代码异味，并结合表4.2所描述的 15 种代码异味，将这 30 种代码异味作为多维安卓特征输入到后续构建的应用程序的安全风险等级预测模型中。

4.3.3　安全风险等级的计算

在建立机器学习模型之前，我们应该先构造一个数据集，知道每个实例的特征和标签。第4.3.2小节描述了从每个实例（即 Android 应用程序）中提取特性的方法。即，我们已经完成了多维安卓特征集的构造。在本小节，我们将详细介绍如何确定每个实例的分类标签（即，确定每一个安卓应用程序的安全风险等级）。我们贴标签的过程包括两个部分。（1）首先，基于开源的 Androrisk 工具，我们将获得应用程序的初始风险分数；（2）然后，我们将该评分结果作为参照，结合人工审核和检验，修改开源工具和评估结果，获得应用程序的最终安全风险等级（图4.9详细说明）。

(1) 安全风险分数的计算

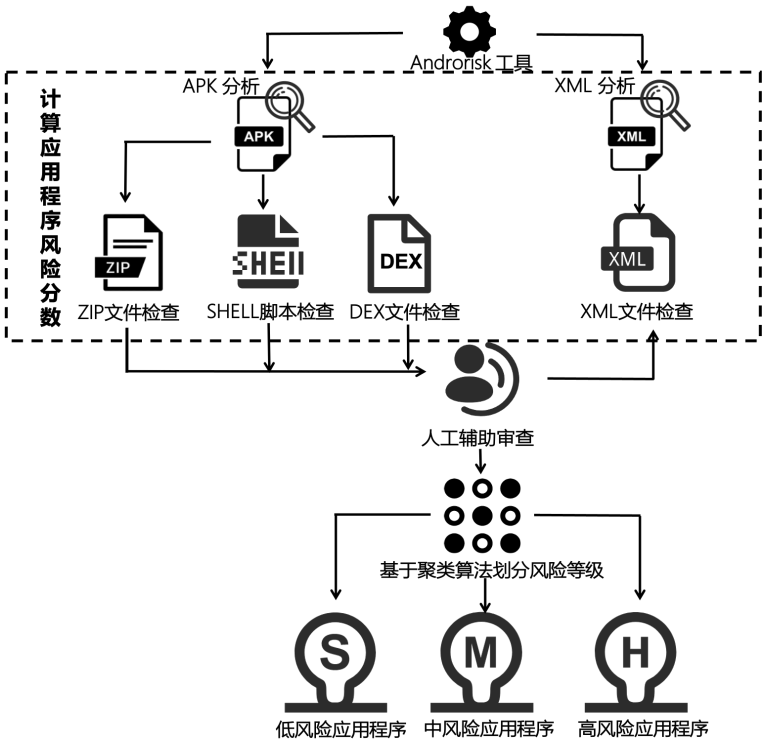


图 4.9: Android 应用程序安全风险等级的计算

在这一章节中，我们首先使用 Androrisk[4]工具以获得应用程序的初始风险分析结果，然后我们手动检查并在必要时修改分析结果，以获得 Android 应用程序的安全风险评分。Androguard 是一个开源的恶意代码检测工具，我们可以使用它去解析 Android 应用程序中的源代码，并从中提取出一些信息，包括 ZIP 文件、APK 文件、DEX 文件、XML 文件等等。Androrisk 是 Androguard 模块之一，是一种广泛应用的安全风险评估工具，其在检测安卓应用程序的隐私和安全性方面的有效性在几篇文献中均有证明 [34, 42]。它有两个安全风险评估模块。这两个模块分别用于分析 APK 文件和 XML 文件。在得到每个模块的分析结果后，我们人工审核了风险结果。通过修改 Androrisk 的代码，修正掉一些假阳性的判断，从而得到更准确的安全风险评分。

对于 APK 文件的分析，Androrisk 认为一些文件的存在会带来安全隐患，因此它会根据文件（包括 ZIP 文件、shell 脚本和 DEX 文件）的存在来计算安全风险评分。在 Androrisk 内部，这些文件被认为很容易被黑客利用。然而，这些文件的存在并不意味着 APK 绝对是有安全风险的。如果 APK 本身有一个预先编码的检查机制来处理这些文件的潜在风险，那么 APK 也可能没有安全威胁。因

[4]https://code.google.com/p/androguard

此，为了避免来自 Androrisk 的假阳性警告，我们手动检查每个 APK，看看它们是否有相应的检查。具体的检查标准如下所示。

ZIP 文件：对于 ZIP 文件，我们主要观察 APK 是否检查了 ZIP 文件名，即查看文件名中否包含特殊的字符串。当文件名中包含特殊字符 "../" 时，我们则认为这是一个危险文件（"../" 可能会导致解压后的文件覆盖其他目录中的文件，导致任意代码的执行）。

SHELL 脚本：对于需要传参的 shell 脚本，我们主要检查 APK 是否对传递给 shell 脚本的外部参数进行了检查。当参数包含分号，&&，|| 等符号时，攻击者可以进行命令拼接注入问题，这样的 shell 脚本被认为是危险的。我们允许传入的参数包括数字、大小写字母、点号、冒号（ipv4 和 ipv6）、斜杆 "/" 和空格。

DEX 文件：在 Android 系统中，classes.dex 负责了项目的核心逻辑业务，因此很多攻击者会通过修改 classes.dex 文件注入恶意行为。对于 classes.dex 文件，我们主要检查 APK 试图加载该 DEX 文件时，APK 是否检查了 DEX 文件的完整性。完整性检验可以通过检查 CRC 值，也可以检查 Hash 值（如清单 4.2 所示）。在我们的研究中那些缺乏必要检查的 apk 才被认为是危险的。

```
1:  // hash值检验和crc检验类似，均是判断其值是否改变
    private void checkCRC() {
3:      // 获取存放在本地的crc加密后的值
        String localCrc = getString(R.string.code);
5:      ZipFile file;
        // 提取出classes.dex文件，计算其CRC值(伪代码)
7:      file = new ZipFile(getApplicationContext().getPackageCodePath());
        ZipEntry entry = file.getEntry("classes.dex");
9:      String str = String.valueOf(entry.getCrc());
        String dexCrc = MD5Util.GetMD5Code(str);
11:     //  判断dex文件是否被修改
        if (!localCrc.equals(dexCrc)) {
            Process.killProcess(Process.myPid());
13:     }
    }
```

清单 4.2: crc 值对 classes.dex 文件的完整性校验

XML 文件：对于 XML 文件的分析，Androrisk 根据每个权限的敏感度，为每个权限分配风险权重（一些可以获取敏感信息的权限，如访问互联网、操纵短信或操纵位置等权限的风险权重更高）。然而，仅仅分析应用程序的敏感权限来评估应用程序的安全性是不可靠的，因为许多良性应用程序因为业务需求，而必须使用一些敏感权限。为了降低风险的误报，我们重新检查了 Androrisk 对权限的分析。更具体地说，我们检查应用程序的功能描述与其申请权

限之间是否存在差异，例如，一个计算器应用程序却申请了位置信息权限（android.permission.ACCESS_FINE_LOCATION），这种不必要的权限被视为安全风险。

Androrisk 工具统计出危险的文件和权限申请，并为每种危险行为匹配一个风险权重，计算出每个应用程序的安全风险分数。然而，在实际代码中，通常会有相应的校验方法，从而规避某些代码漏洞。因此，Androrisk 工具存在一定的多报现象，造成计算出的应用程序的安全风险分数偏高。我们以该工具的初始评判结果作为参考，在源代码中加上对上下文的判断，修正某些过度检测的文件数，从而得到相对公正的安全风险分数。

(2) 安全风险等级的计算

根据第4.3.3节，我们获得了单个应用程序的安全风险分数。对于安卓应用程序的相关从业者而言，一个应用程序的风险分数依然不能让他们直观清晰的了解到程序是否处于安全状态。因此，我们决定将连续型的风险分数离散化为安全风险等级。这将带来几点好处：（1）离散化的安全风险等级相对于连续型的安全风险分数更易理解。假设两个应用程序的安全风险分数分别为 53.3 和 81.1，从连续型特征来看，这些应用处于高风险或是低风险，还需要通过比较其他风险分数才能了解。但将其转换为离散型数据后（低风险、高风险），则可以一眼看出这些应用程序是否处于高风险。（2）对于某些分类学习算法而言，它们要求数据标签为离散型数据。有效的离散型数据相对于连续型数据来说，能加快模型的拟合速度，提高算法的效率和模型的抗噪能力。（3）对于一些存在瑕疵的连续型数据集，某些异常的数据点会影响模型的训练。而离散型数据可以忽略某些数据中的杂点，使模型的拟合结果更加稳定。

基于以上原因，我们进一步离散化这些连续的安全风险分数，以得到它们的安全风险等级作为它们的最终分类标签。为了将连续风险评分离散化，我们将 K-means 聚类算法应用于 Android 应用程序的安全风险分数。K-means 聚类方法有两个输入参数：（1）簇的数量 k（2）对象的数量 n。在我们的研究中，我们总共收集了 n = 4575 个 Android 应用程序作为创建集群的数据，并通过使用 Elbow 方法 [110] 来确定簇的数目 k。Elbow 方法又称肘部法则，它通过对 k 设置一系列的值（1,2,3,4,...n），在数据集中运行 k-means 聚类，直至模型逐渐收敛，聚类效果达到最佳。在 k-means 聚类方法中，它以最小化误差平方和（SSE）为目标，度量聚类效果的好坏。对于 n 个点的数据集，Elbow 方法依次对 k 赋予不同的值（1,2,3,4,...n），并利用 k-means 聚类，在聚类完成后计算此时的误差平方和，观察此时聚类效果。随着 k 值越靠近正确值，误差平方和也会逐渐变小。考虑极端情况，当 k = n 时，其误差平方和等于 0，此时每个点都是其所属簇的中

心点。当 k 小于实际簇类数时，k 的增加会显著增加簇内数据点的凝聚力，此时 SSE 的值将会显著降低。当 k 值越来越接近实际簇类数时，聚合度的增量会随着 k 的增加而迅速下降。随着 k 值的增加，SSE 的跌幅将减缓，最终收敛。通俗来说，SSE 和 K 之间的变化关系就像一个肘部，在下降速度突然减慢的地方形成拐点。此时 K 被认为是数据集的最佳的簇类数。误差平方和（SSE）指数越低，聚类效果越好。计算 SSE 的方法如公式4.1：

$$\text{sse} = \sum_{i=1}^{k} \sum_{x \in C_i} \text{dist}(x, O_i)^2 \tag{4.1}$$

其中，k 表示簇类数，O_i 表示第 i 个簇 C_i 的簇心，其计算公式如4.2所示。$\text{dist}(x, O_i)$ 表示数据点 x 和 O_i 的相似度，相似度一般采用的是欧几里得度量见公式4.3。

$$O_i = \frac{\sum_{x \in C_i} x}{\|x\|} \tag{4.2}$$

$$d_{(x_i, x_j)} = \sqrt{\left(x_i - x_j\right)^{\mathrm{T}} \left(x_i - x_j\right)} \tag{4.3}$$

在本文中，我们总共收集了 4575 个开源 Android 应用程序。为了划分出每个应用程序的安全风险等级，我们使用 elbow 方法，确定了集群的数量。如图4.10所示，我们计算了 K 在 1 到 8 之间的 SSE 值，发现当 K 在 1 到 3 之间时，SSE 的变化最大，这意味着将这些安卓应用程序划分成 3 个簇时，聚类效果是最好的。

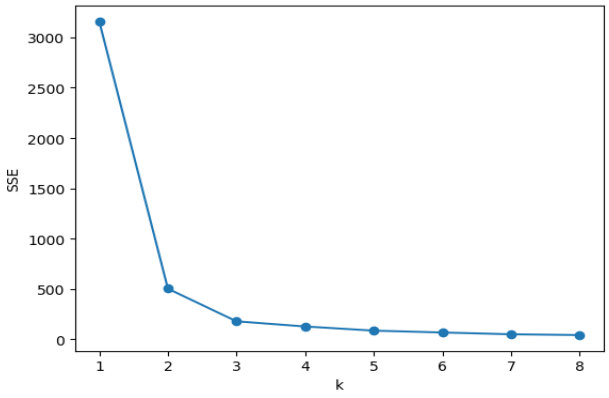


图 4.10: k-means 的最优 k 值计算

因此，我们将 k 设为 3，并使用 K-means 将 4575 个应用程序分成 3 个集群。我们给这三个集群分配了三个安全风险标签，即低风险（L）、中风险（M）和高

风险（H）。三个簇的质心分别为 0.15715467、50.90121676 和 92.89716516。低风险、中等风险和高风险的 Android 应用程序数量分别为 1503、2633 和 439。

4.3.4　特征指标的预处理

如4.3.2所描述，当我们获得了单个实例（Android 应用程序）的原始特征指标（即 Java 静态代码指标和 Android 代码异味）和类标签（安全风险等级）后，我们就可以在它们的基础上建立一个机器学习模型。但是，考虑到本研究中提取的特征值在尺度上存在较大差异，并且这些特征指标之间有多重共线性问题，我们决定先对这些特征进行预处理，以避免这些因素对模型的构建产生一些负面影响。此外，我们还观察到我们的数据集是非常不平衡的，即，不同类的实例数是不同的，这种不平衡问题也应该在建模时解决，因为它会极大地影响分类模型的训练。在完成以上步骤之后，我们可以应用典型的机器学习算法来建立预测模型。具体的算法和理论知识见第三章特征预处理章节。

(1) 特征归一化

在我们的研究中，我们的特征指标的数值（即 Java 静态代码指标和 Android 代码异味）在它们的范围尺度上是完全不同的。例如，一些 Android 代码异味的指标大小在 0 到 10 之间，而一些 Java 静态代码指标，如代码行指标则高达数千。如果不处理尺度问题，预测结果可能会被一些值较大的特征所支配，也会影响个体特征的合理比较和模型的训练速度。为了克服以上问题，我们决定将特征指标转换成无量纲化变量，即完成特征的标准化。特征的标准化通常有两种方法。一种叫做最小-最大标准化，它将所有特性大小限制在 0 到 1 的范围内。另一种叫做零均值标准化 (z-score 标准化)，它将原始数据缩放到统一度量标准。在本文中，我们采用的就是零均值标准化方法。采用该方法后，各维度特征的标准差为 1，平均值为 0。这可以帮助我们避免预测结果被一些具有较大值范围的特征所支配的问题。对于每个应用程序，我们应用了 *z-score* 标准化来处理其特征指标（即 Java 静态代码指标和 Android 代码异味）。在 *z-score* 标准化后，每个特征的值即落在了有限范围内。

(2) 特征独立化和特征降维

一般来说，一些特性实例可能与其他特性实例相关（我们的例子也是如此）。例如，Android 应用程序的代码行数越多越容易造成代码的高复杂度。如果一个预测模型不能完全处理特征之间的多重共线性问题，可能会增加因变量的可变性，从而降低模型的性能。本文采用主成分分析法（PCA）来解决特征之间存在依赖性的问题。PCA 将原始特征指标重新组合，通过赋予指标不同的权重，生成新的特征指标，这类特征之间没有关联。利用主成分分析（PCA），不仅可以

降低特征维数，还消除特征之间的依赖性，从而获得更具竞争力的预测模型。应用 PCA（主成分分析）算法以后，21 个 Java 静态代码指标和 30 个 Android 代码异味指标被降低到 20 维，降维后每个特征向量由原始特征指标而组成（赋予不同的权重）。

(3) 类不平衡处理

在我们的研究中，不同的安全风险等级的实例数量是极不相同的。在本文中，我们总共采样了 4575 个开源的 Android 应用程序，其中安全风险等级为 L（低）的有 1503 个，安全风险等级为 M（中）的有 2633 个，安全风险等级为 H（高）的有 439 个。由此可见，具有高风险等级的样本数量非常小，而具有中风险等级的样本数量非常大，这就是所谓的类不平衡。不考虑类不平衡问题的话，学习算法往往只会学习到多数类样本信息，造成训练结果的不准确。例如，考虑极端情况，假设有 98 个多数类和 2 个少数类。在一个以最小化错误率为目标的学习算法中，它可以将所有的实例都分类到多数类中。这样，该训练模型的准确度依旧很高，可以达到 98%，但是所有的少数类实例都会被错误地分类。为了得到一个合理的预测模型，必须仔细处理类不平衡问题。

在机器学习领域，有很多方法都可以被用于解决类别失衡的问题。其中，最简单有效的方法是过采样和欠采样。过采样试图通过添加少数类实例的多个副本来平衡类；而欠采样则试图通过丢弃多数类的样本实例来平衡类。在本次研究中，我们决定摒弃欠采样方法。最主要的原因是原始训练数据量小（三个安全风险等级中的应用程序都很少）。而欠采样方法会丢弃一些多数类实例，使得我们的训练数据量变得更少，模型训练数据不充分，无法建立一个好的预测模型。而过采样方法则是通过简单地重复少数类来实现样本类间的平衡。它本质上并没有为模型引入更多的有效信息，相反会为模型带来噪音数据。因此对于过采样方法，我们没有使用原始的随机过采样方法，相反，我们使用了一种非启发式方法，它在过采样的基础上进行了优化，即 SMOTE 算法 (合成少数类过采样技术)[108]，以 ENN 算法 (Wilson’s edited nearest neighbor rule)[111] 来解决类不平衡问题（SMOTE 方法根据少数类样本线性插入新样本，而不是通过简单复制样本以扩充数据集，这可以帮助我们避免过拟合问题。基于 K-最近邻（KNN）算法，ENN 方法可以删除任何被错误分类的样本，解决 smote 引入的噪声数据以及样本空间被侵占的问题）。

具体来说，在 4575 个开源 Android 应用程序中，它们原始的安全风险等级分布是 1503 个应用程序处于 L（低）风险等级、2633 个应用程序处于 M（中等）风险等级和 439 个应用程序处于 H（高）风险等级。在应用 SMOTE + ENN 算法平衡三类安全风险等级后，属于低风险（L）、中风险（M）、高风险（H）的应

用程序的数量分别为 1406、757 和 2195，不同类别之间的样本数差异明显降低。这说明了 SMOTE + ENN 算法在处理类不平衡问题上是有效的。

4.3.5　预测模型的构建

在预测 Android 应用程序的安全风险等级时，不同的机器学习方法所构建的预测模型会有不同的性能。为了理解 Android 代码异味在预测应用程序的安全风险方面的作用，同时构建出预测性能最佳的模型。我们将 Java 静态代码指标和代码异味作为安卓特征指标，并应用九种流行的机器学习算法，来预测 Android 应用程序的安全风险等级。我们在研究中应用了九种典型的机器学习算法：朴素贝叶斯[5]（Naive Bayes，NB）、支持向量机[6]（Support Vector Machine，SVM）、k-最近邻[7]（K-NearestNeighbor，KNN）、决策树[8]（Decision Tree，DT）、梯度提升树[9]（Gradient Boosting Decision Tree，GBDT）、随机森林[10]（Random Forest，RF）、逻辑回归[11]（Logistic Regression，LR）、卷积神经网络[12]（Convolutional Neural Networks，CNN）以及多层感知器[13]（Multi-Layer Perceptron，MLP）。因为这九种算法均是常见的机器学习算法，具体细节不在文中赘述，详见所附链接。

4.4　故障数计数模型

在本章节中，我们将讨论代码异味对安卓应用程序质量的影响。具体来说，我们研究了安卓代码异味与应用程序故障数之间的关系，故障数统计了应用中有关安全，内存，性能等各项问题和故障数量。我们计算了代码异味和故障倾向性的关系（是无关联还是正关联还是反关联还是组合关联），从而说明哪些代码异味是开发者在开发时无需关注的，哪些代码异味是应该特别留意与修复的。

4.4.1　故障数的统计

在软件测试和开发领域，有很多概念被用于描述软件中的问题：（1）fault（故障）描述软件中程序设计的不正确（2）vunlerability（脆弱点）描述软件中的

[5]https://en.wikipedia.org/wiki/Naive_Bayes_classifier

[6]https://en.wikipedia.org/wiki/Support-vector_machine

[7]https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

[8]https://en.wikipedia.org/wiki/Decision_tree

[9]https://en.wikipedia.org/wiki/Gradient_boosting

[10]https://en.wikipedia.org/wiki/Random_forest

[11]https://en.wikipedia.org/wiki/Logistic_regression

[12]https://en.wikipedia.org/wiki/Convolutional_neural_network

[13]https://en.wikipedia.org/wiki/Multilayer_perceptron

静态缺陷（3）error（错误）描述运行存在缺陷的软件代码后，导致的错误结果（4）failure（失效）描述软件问题的外在现象，用户所能观察到的软件失效行为（5）bug（问题）表示对软件失败和错误的一般描述（6）defect（缺陷）的存在可以引起多种 Bug。在本文中，我们不会对软件问题有这么详细的区分。我们将以上所有的软件问题统称为故障，它会影响软件的各方面，包括性能、安全、内存、可移植性和效率等，衡量了一个软件产品的质量。我们应该识别并修复这些故障，以提高 Android 应用程序的健壮性。

以前的研究人员都使用数据仓库来收集故障数据集。用于收集软件故障的数据仓库大致可分为三种类型：私有/商业仓库、部分公共/自由仓库和公共仓库[112]。大多数研究人员使用公共数据仓库（如 Bugzilla、JIRA、PROMISE）来收集故障数据集，这有利于模型的再现性和各种模型的比较分析。Hall 等人 [113] 评估了从公共数据仓库中提取故障数据的三种方法。他们发现 Zimmermann 等人 [114] 提出的方法从召回率和精确度来看不如人工方法好。然而，人工方法耗时费力，在研究大型数据集时，大多数研究人员更愿意使用 Zimmermann 等人 [114] 提出的方法来收集故障数据。Zimmermann 等人 [114] 主要通过关键字位置来识别软件故障，然后使用 Bugzilla 故障报告来确认这些故障。

相对于一般的应用软件而言，Android 应用程序依常赖于第三方库、开发规模小、更新迭代快，目前还没有可用于 Android 应用的公共数据仓库。而且，大多数的研究都致力于极少数的开源项目上，其覆盖范围有限，研究结果可能带有偶然性。为了研究更多的开源安卓应用程序，我们决定在 GitHub 上收集开源应用，并利用 GitHub 上的 commit 规范，定位出有关故障修复的提交，最后通过人工检查来验证这些提交是否真的是故障修复。在应用程序的一个版本开发周期内，我们在”open” 和”closed” 的问题中找出包含所有故障关键字的提交，并统计出应用程序的故障数。这种方法总结如下。

1) 在已提交和已解决的问题中找到包含 “fault”、“vunlerability”、“error”、“failure”、“defect”、“bug”、“mistake”、“fix（ed）” 和 “update（d）” 关键字的提交。特别注意的是，在匹配过程中，关键字的大小写被忽略。

2) 由于开发人员的不规范提交，很多带有故障关键字的 commit 并不一定真的是故障修复，因此，对于成功匹配关键字的 commit 还需要通过人工进行复查，判断其是否真的是故障修复。由于故障较多，我们计划采用随机抽样检测。

GitHub 上的 Android 应用程序是各不相同的。有些 Android 应用程序有很多的开发版本，有些只有一个版本，而有些甚至没有一个版本。为了更合理地收

集故障数据，我们决定在安卓应用程序的一个版本开发周期内，根据关键字匹配技术，统计相应的故障数量。根据 Android 应用程序的版本数，我们进行了如下的处理。

1) **无版本：**对于没有版本的 Android 应用程序，我们从最初提交的 commit 开始检索，找到第一个完整且可运行的程序，并记录此次 commit 与最新 commit 之间的故障数。

2) **一个版本：**对于只有一个版本的 Android 应用程序，我们记录从这个版本到最新提交的 commit 之间的故障数。

3) **两个及以上版本:** 对于有两个或两个以上版本的 Android 应用程序，我们记录最新版本和次新版本之间的故障数量。

4.4.2　原始数据的收集

本文从 GitHub 网站上收集了 4575 个基于 Java 语言编写的开源安卓应用程序，并根据故障关键字统计了每个应用程序的故障数（详见4.4.1）。为保证故障数与应用程序的一一对应，在统计故障数时，本文以一个开发版本作为计数单位，计算每个应用程序在一个版本周期内包含的故障总数。在原始的数据集中，一个应用程序中包含了多个开发版本。根据4.4.1节所述的版本处理方法，我们对每个应用程序只选定了一个版本进行研究。至此，4575 个原始应用程序，删减为 645 个应用程序。我们随机选取其中的 516 个应用程序作为训练数据，另外 129 个应用程序作为预测数据。由图4.11所见，原始应用程序中的故障数不服从正态分布，离散程度较大，且大部分的应用程序的故障数为 0。

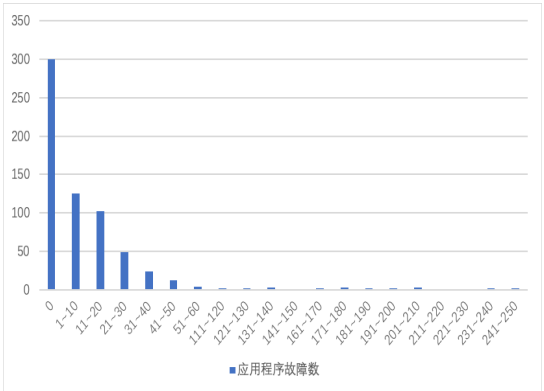


图 4.11: 应用程序的故障数统计

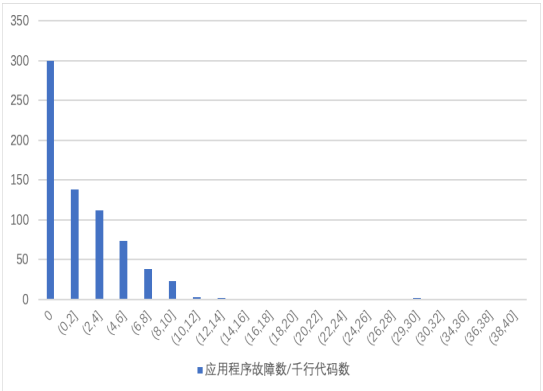


图 4.12: 应用程序的故障数密度统计

4.4.3　故障数模型的构建

本文研究了故障数与 Android 代码异味之间的关系。其中，故障数涵盖了安全、性能、内存等各项风险问题。它是一个非负整数，是通过计数方式来获得的。计数模型是一种用于做数据分析的线性模型，其中响应变量是计数类型。泊松回归模型和零膨胀泊松模型是分析计数数据的经典离散模型。泊松分布的一个重要先决条件是均值等于方差。方差一般指示数据的离散程度。在实验数据中，方差通常较大，而均值通常较小。此外，泊松分布要求观测值是独立的。然而在重复实验的过程中，容易造成数据间的依赖，因此必须考虑组内的相关性。

许多统计分析技术都要求数据是呈正态分布的。在确定合适的计数模型算法之前，我们观察了原始数据的分布。有如下三个特点。（1）图4.11显示了我们的因变量，即文件中的故障数，它不呈现正态分布，且离散程度大。（2）此外，在 645 个应用程序中，绝大多数的应用程序都不包含任何故障，即，很多的应用程序的故障数等于 0。产生上述现象的原因主要是安卓应用程序的开发周期短，迭代速度快，且在 GitHub 上的开源应用程序多为中小型项目，在一个项目上没有完整的开发者团队，对应用程序的测试不完善。因此，我们针对一个版本周期内收集的故障数较少，很多都是 0 故障。（3）最后，我们对数据进行了过离散实验（验证其方差是否大于均值）。原假设是数据的方差等于均值。我们运用 R 语言中的 dispersiontest( ) 函数对故障数进行过离散检验。结果发现，因为偶然性而造成的数据的方差等于均值的概率较小（$p < 0.001$），不能接受原假设。即，实验中的故障数据集不呈正态分布，方差大于均值，且存在零膨胀的现象。在此类实验数据上，与泊松回归模型相比，负二项回归和零膨胀回归对数据的拟合程度更高。

此时，我们有两种解决方案如下所示。

（1）使用故障密度（即文件中每行代码（NCLOC）的故障数），而不使用故障数量作为响应变量。探究大文件的应用程序是否包含更多的故障数，用于实验的故障数据的分布是否更正常。我们以应用程序中每千行代码的故障数 (faults/K-LOC) 作为故障指标，衡量应用程序的质量。该故障指标又称为故障密度 (Density of Faults)，其测量单位是 faults ／ KLOC。故障密度 = 故障数量/千行代码的数量。结果如图4.12所示，故障密度的分布依旧不是正态分布。此时，由于缺乏呈正太分布的故障数据，导致我们不能使用很多的计数算法（如泊松分布）。

（2）使用其他计数算法。本文采用了两种回归模型来分析安卓应用程序的故障数与安卓代码异味之间的关系。即，负二项回归模型 (NB) 和零膨胀负二项回归模型 (ZINB)。这两种回归模型可以允许响应变量存在过度离散分布，即，它允许故障数的方差超过平均值。另外，通过统计我们发现，645 个安卓应用程序

中有 300 多个应用程序的故障数为 0，故障数等于零的组占比将近 50%。因此对于这类数据，如果我们采用是普通的泊松回归模型，那么数据中的大量零值会带来方差和均值的差异，其预测概率将会产生较大的偏差。零膨胀负二项回归模型则解决了数据中过多的应用程序的故障数为“0”的问题。本文以某种代码异味作为自变量，应用程序的故障数作为因变量，应用 NB 和 ZINB 算法建立了计数模型，并根据 AIC 和 BIC 指标进行对比分析，选出了最佳模型。

4.5　本章小节

本章重点介绍了本文提出的移动应用风险评估技术，构建了移动应用的安全风险等级预测模型和故障数计数模型，分别用于评估应用的隐私和安全性风险以及应用的质量风险。基于现有文献和研究，我们将一些代码的坏结构定义为代码异味，自定义了 15 种全新的安卓代码异味，并提出了相应的修复意见。依据这些定义，我们开发了自己的代码异味检测工具，用于检测出哪些代码中存在代码异味。通过定义安卓代码异味，将安卓应用程序中的特殊语法和使用特征化和量化。在具体定义完代码异味后，我们收集来自 GitHub 上的开源安卓应用程序，提取安卓特性指标（安卓代码异味和 Java 静态代码指标），并对指标进行预处理，使得数据更平滑降噪。通过结合不同的机器学习分类算法，我们构建了几种安卓应用的安全风险等级预测模型，帮助大家识别出高风险的应用程序。此外，我们还选取了其中的五种代码异味，应用不同的回归算法（负二项回归和零膨胀负二项回归），建立了故障数的计数模型。我们探索了故障数与具体的 Android 代码异味之间的关系，研究了代码异味对应用程序的安全，内存，性能等各方面的影响，从而提醒开发人员在开发过程中需要注意修复哪些代码异味。

第五章　实验设计与分析

5.1　实验一：DACS 工具的评估

如第四章第4.2所描述，本文自定义了 15 种全新的代码异味，根据这些代码异味的定义，DACS 工具实现了不同的异味检测算法。在将 Java 开源代码转换成抽象语法树之后，DACS 工具基于检测算法不断搜索抽象语法树，从而找出哪些代码片段包含了代码异味。

5.1.1　实验目的

为了保证 DACS 工具在检测自定义的 15 种代码异味时的有效性，我们设计了一个实验用于是评判 DACS 检测源代码中代码异味的能力。具体来说，我们的研究问题如下：

问题一：DACS 工具在检测 Android 代码异味方面是否和人工检测结果保持一致，工具的检测结果是否是可靠的？

问题一的回答可以直观表现出 DACS 工具的检测性能，为后续的风险评估框架提供有力保障。由于该工具检测的代码异味是我们自定义的，目前尚未有成熟的检测工具与我们的工具进行比较。为了让 DACS 工具的检测结果有可靠的参照物，我们邀请了人工完成相同应用程序的检测评判工作，并将两份结果进行一致性计算。特别注意的是，人工检测与 DACS 工具只共享相同的代码异味定义，不共享其他任何检测策略。

5.1.2　实验数据

这项研究的实验数据是由一组 20 款开源的 Android 应用程序组成，这些应用程序均是 github 上星级较高的中小型开发项目，它们属于不同的类别，具有不同的大小和开发者。实验中涉及的安卓应用程序的详细信息如下表5.1所示。

表 5.1: DACS 工具自动化检测应用程序集

ID	应用名	类型	版本	地址
1	NewPipe	影音	v0.14.2	https://github.com/TeamNewPipe/NewPipe
2	FxcnBeta	阅读	3.2	https://github.com/ChaosLeung/FxcnBeta
3	RGB-Tool	工具	v1.4.2	https://github.com/fasteque/rgb-tool

4	EasyBudget	生活	1.6.2	https://github.com/benoitletondor/EasyBudget
5	PodTube	视频	1.3.2	https://github.com/ghostwan/PodTube
6	Typesetter	工具	v1.0.1	https://github.com/bignerdranch/Typesetter
7	Snapdroid	影音	v0.15.0	https://github.com/badaix/snapdroid
8	LAY	健康	v0.1.3	https://github.com/florentphilippe/LAY
9	Habitat	工具	v1.0.9	https://github.com/ardevd/habitat
10	WhuHelper	学习	v1.0	https://github.com/zfb132/WhuHelper
11	Cupboard	生活	1.0	https://github.com/SCCapstone/Cupboard
12	RITA	工具	v1.0	https://github.com/brenov/rita
13	Desserter	学习	v1.0	https://github.com/drulabs/desserter
14	TorchLight	购物	v2.4	https://github.com/JavaCafe01/TorchLight
15	VINCLES	社交	3.12.0	https://github.com/AjuntamentdeBarcelona/vincles-android
16	intra42	学习	v0.6.2	https://github.com/pvarry/intra42
17	Natibo	学习	v0.2.4	https://github.com/chickendude/Natibo
18	Lexica	游戏	v0.11.4	https://github.com/lexica/lexica
19	Timber	影音	v1.6	https://github.com/naman14/Timber
20	MateSolver	游戏	v2.6	https://github.com/streblow/MateSolver

5.1.3　实验设置

为了回答问题一，我们在实验选取的 20 款应用程序上运行了 DACS 工具。为了评估它的性能，我们邀请某大学的一名硕士研究生手动分析所考虑的应用程序，以便统计每种代码异味的数量。基于 15 种代码异味的定义，该学生手动分析了 Andoroid 应用程序的源代码，寻找包含异味的代码。整个过程大约需要 72 个小时。然后，第二个硕士生重新分析包含异味的代码，以检验第一个学生识别出的包含异味的代码是否存在误判。两位学生之间在做实验时是独立且无交流的。结果表明，在第一个学生分类为包含异味的代码中，只有 11 个代码片段被第二个学生归类为误判。经过两个学生的讨论，这 11 个代码片段中有 6 个被确定为误判，随后，这 6 个误判结果被最终更正。此外，为了避免偏见，学生们不知道实验细节和 DACS 工具用来检测代码异味的具体规则算法。至此，一份相对公正准确的代码异味检测结果产生，我们认为其是一个比较基准。

在人工完成代码异味检测工作后，我们挑选出其中包含代码异味的.java 文

件，将这些文件作为受试者，供 DACS 工具进行后续的评估检测工作。我们将 15 种代码异味作为检测对象，并在 DACS 工具上运行挑选出的.java 文件集，由 DACS 工具根据其实现的异味检测算法，计算每个文件中的每种代码异味的数量，其中 0 表示该代码异味不存在，>=1 的整数表示该代码异味存在。针对 15 种代码异味，我们对挑选的文件集进行依次检测，得到一份人工检测结果，以及一份 DACS 工具的检测结果。它们的结果都是连续型的数值，表示每种代码异味在 class 文件中的数量的累加值。

5.1.4　评价指标

目前没有一组开源且详细的代码异味数据可用，因此，我们无法像 Eclipse 这样的开源项目一样能评估出任何工具的性能，无法绝对确定工具在此类真实系统上是否能准确检测到代码异味。在本文中，由于缺少可靠的检测数据可以与工具的检测结果进行比较，因此我们不使用精确性或召回率作为评价指标。在本文中，我们将人工检测结果作为参考标准，评估 DACS 检测结果的准确性。测试者之间是相互独立互不影响的，因此，对于每种代码异味，我们需要使用某评价指标来分析了 DACS 工具和人工检测之间的一致性。Cohen 的 Kappa 统计量是常见的用于一致性检验的指标，然而它只能应用于分类评级，不适用于我们的计数型工具的评估。当我们的评分是连续性整数时，Lin’s 的一致性相关系数 (CCC) 则可以用来衡量两个评估者的一致性 [115]，即比较同一变量的两个测量值的差异。假设有两个随机变量 x 和 y，则一致性的相关系数 ρ_c 的公式定义为5.1。

$$\rho_C = \frac{2\rho\sigma_x\sigma_y}{\left(\mu_x - \mu_y\right)^2 + \sigma_x^2 + \sigma_y^2} \tag{5.1}$$

其中，μ_x 和 μ_y 是两个变量的平均值，σ_x^2 和 σ_y^2 是相应的方差。ρ 是两个变量之间的相关系数。当我们的数据集的长度为 n 时，即，N 份数据对 (x_n, y_n)，$n = 1, \ldots, N$。那么 CCC 的计算公式为5.2。

$$r_c = \frac{2s_{xy}}{s_x^2 + s_y^2 + (\bar{x} - \bar{y})^2} \tag{5.2}$$

其中，平均值计算如公式5.3所示，方差计算如公式5.4所示，协方差计算如公式所示5.5。

$$\bar{x} = \frac{1}{N}\sum_{n=1}^{N} x_n \tag{5.3}$$

$$s_x^2 = \frac{1}{N}\sum_{n=1}^{N}(x_n - \bar{x})^2 \tag{5.4}$$

$$s_{xy} = \frac{1}{N}\sum_{n=1}^{N}(x_n - \bar{x})(y_n - \bar{y}) \tag{5.5}$$

在本文实验中，我们有 15 种自定义的代码异味，即，我们的有 N=15 份数据对 (x_n, y_n)，$n = 1, \ldots, 16$。数据对表示针对每一种代码异味 i，人工检测的数量 x_i，以及 DACS 工具的检测数量 y_i。

和相关系数一样，$-1 \le \rho_C \le 1$ 并且 $-1 \le r_C \le 1$。当指标越接近 +1 时，表示 x 和 y 之间有强一致性，当指标越接近-1 时，表示 x 和 y 之间强不一致性，当指标越接近 0 时，表示 x 和 y 之间的不一致性。对于如何解释指标的其他值，目前尚未有明确的普遍意见。不过，还有一种公认方法是将 Lin 的 CCC 指标解释为斯皮尔曼 (Spearman) 的相关系数（例如，小于 0.20 的一致性较差，而大于 0.80 的一致性较好）。在本文中，我们也沿用这种解释，来分析 DACS 工具和人工检测结果的一致性强度，具体解释细节见表 5.5。对于 Lin's 一致性相关系数的计算，我们基于 R 语言，使用 DescTools 库中的 CCC 函数，并输入两份代码异味检测结果，计算组内的相关系数。CCC 函数会返回 Lin 的一致性相关系数，并输出显著性水平等于 α 的 1-α 置信区间（α 默认为 0.05）。

5.1.5　实验结果分析

在 20 个实验应用程序中，DACS 工具总共检测到 322 个代码异味实例（平均每个应用程序有 16 个代码实例包含异味）。最常见的是弱加密算法（WCA）异味（163 个实例）、不安全的随机数（UR）异味（61 个实例）和数据备份（DBA）异味（17 个实例）。在这 20 个开源应用程序中，总共包含了 2232 个.java 文件，其中包含代码异味的.java 文件集合的个数为 283 个。我们以 15 种自定义的代码异味作为检测对象，统计这 20 个应用程序中每种代码异味的代码实例（实例粒度为方法级）数。对比了工具检测结果和人工检测结果的一致性。

如表5.2和表5.3所示,我们的DACS工具和人工在检测证书验证不正确(ICV)异味，Intent 攻击（ISU）异味和数据备份（DBA）异味时完全一致，在检测其他代码异味时，结果差异也不大（差值保持在 5 以内）。此外，一个有趣的现象是，DACS 工具的检测结果大部分都比人工检测的结果大，这就意味着工具存在一定的多报现象。这是因为人工在检测过程中，会结合代码上下文，判断开发者是否有对一些危险行为进行处理，当有对应的处理方法时，即使代码中包含了

易受攻击的片段，人工也不会认为这是一个代码异味。而工具的检测则单纯匹配检测规则，不够灵活和变通，因此存在一定的误判情况。

表 5.2: 代码异味的检测结果 I

代码异味	WCA	ICV	UCE	ISU	HA	EC	WRCE	WSP
人工结果	158	2	1	2	12	2	4	3
工具结果	163	2	2	2	8	3	5	5

表 5.3: 代码异味的检测结果 II

代码异味	WDNS	DBA	GFRW	PRW	MU	SDV	UR
人工结果	6	17	15	8	5	13	57
工具结果	8	17	16	11	6	9	61

总体来说，针对这 15 种 Android 代码异味而言，DACS 工具能够正确识别 Android 应用程序中几乎所有的代码异味实例。如表5.4所示，Lin's 相关一致性系数 r_c 等于 0.9980264，置信度 0.95 上的置信区间是（99.5%,99.9%），这表明工具的检测效果和人工检测结果的一致性程度很高。只有在两种情况下，结果不能达到理想情况，会存在代码异味的漏检，即恶意解压缩（MU）异味和头文件（HA）异味。

表 5.4: 人工检测结果和 DACS 检测结果的 Lin's 一致性

est	lwr.ci	upr.ci
0.9980264	0.9951825	0.9991921

为了理解结果背后的原因，我们再次解析了这些代码异味，发现了 DACS 工具漏掉了一些实例，因为受恶意解压缩（MU）代码异味影响的类与检测规则中考虑的类使用了不同的压缩库。例如，在 VINCLES 应用程序（id=15）中，在 com.iceteck.silicompressorr 包中定义了 SiliCompressor 类。这个开源库 SiliCompressor[1] 用于对视频、语音等文件（GallerypPresenter 类，ChatPresenter 类和 FileUtils 类）进行压缩处理，但是 DACS 工具目前尚未引入 SiliCompressor 库，因为此类压缩代码将不会被 DACS 工具所检测。同时，造成头文件（HA）异味漏报的原因也大体如此。Android 应用程序请求网络通讯时，常用的 http 请求库除了

[1]https://github.com/Tourenathan-G5organisation/SiliCompressor

HttpClient、OKhttp 外，还有更高层次的封装如 Feign、Retrofit 和 Volley 等等。在目前的检测算法中，DACS 只匹配了一般的网络请求库，因此检测器无法正确识别某些设计缺陷。

虽然在本例中，DACS 工具会遗漏某些 MU 代码异味和 HA 代码异味，但是在 DACS 工具的后期迭代更新中，我们将考虑引入 Android 开发人员使用的各种类型的库。此外，这个问题提供了一个潜在的方法来提高检测工具的性能。随着对其他第三方库的支持，该工具的性能将更高。

问题一的回答：总体而言，除了对 MU 异味和 HA 异味的检测存在漏检现象，对大部分代码异味而言，DACS 工具的检测结果和人工检测的结果保持强一致性，其 Lin's 相关一致性系数 r_c 能达到 0.99，证明了我们的工具能够有效地检测出自定义的 15 种代码异味，捕获移动开发人员在应用开发时的编程问题。最后，我们可以确定的是，就检测准确性而言，DACS 工具是有效的。

5.2　实验二：安全风险等级预测模型的建立与评估

如第四章第4.3所描述，为了评估应用程序的隐私和安全性，本文建立了安全风险等级预测模型。本文首先提取了代码异味和 Java 静态代码指标作为多维安卓特征，并对特征进行预处理，使得处理后的特征平滑降噪。接着计算了应用程序的安全风险等级作为标签。最后应用机器学习分类算法，构建了应用程序的风险等级预测模型。

5.2.1　实验目的

本实验的目的是评估在真实的安卓应用程序中，模型在预测安全风险等级方面的能力。为了能更直观的反映该模型的性能，我们应用了九种流行的机器学习算法，并组合不同的安卓特征指标，旨在构建性能最佳的风险预测模型。同时，为了更好地了解代码异味在风险评估工作时的作用，我们将 Java 静态代码指标作为安卓特征构建的模型当做本实验的对比模型，比较模型在引入代码异味后是否有性能上的提升。

本文主要回答了以下三个研究问题。

问题一: Java 静态代码指标和 Android 代码异味之间的相关性如何？安卓特征指标之间是否是相互独立，互不依赖的？

问题二：我们构建的安全风险等级预测模型的性能如何？安卓代码异味是否真的能提高模型的风险预测能力？

问题三：在预测 Android 应用程序的安全风险等级时，哪些特征指标更重要？具体哪些安卓代码异味的引入会造成应用程序的高风险？

问题一旨在了解 Java 静态代码指标和 Android 代码异味之间是否存在交互关系。通过计算特征指标之间的相关系数，了解哪些指标之间存在联系。进一步探索影响 Android 代码异味的因素。在分类模型中，特征之间要独立化，否则会引入冗余或干扰信息。

问题二的目的是调查所提出的指标，特别是安卓代码异味是否对安全风险等级的预测有用。尽管在之前的研究中，Android 代码异味在模型中表现出重要的影响力，但有必要检查它们是否能够在更一般的应用程序中表现良好。此外，问题二是研究风险预测模型的影响因素。为了避免算法及参数对预测性能的影响，我们选择多种机器学习方法，分别使用 PCA 算法或 SMOTEENN 算法建立预测模型，并通过绘制 ROC 曲线直观评价预测模型。

问题三的提出是为了研究对于所有的特征指标而言，它们在风险预测模型中是否具有相同的重要性，特别是研究 Android 代码异味对应用程序的影响。出于实际原因，我们可以给应用开发人员一些指导，让他们知道应该哪些代码异味会造成应用程序的高风险，以便在日后开发中知道需要着重修复哪些代码异味，避免哪些不良的代码结构。

5.2.2　实验数据

如第4.3.1节所描述，本实验的原始数据是来自于 Github 上的 4575 个开源安卓应用程序。这些应用程序的类型和大小不尽相同，但均是以 Java 语言开发的应用程序。如第4.3.2节所描述，本实验利用自动化工具，提取出了多维的安卓特征，包括了 Java 静态代码指标和安卓代码异味。在对这些特征指标进行预处理之后（详见4.3.4节），输入不同的机器学习算法（详见4.3.5节），构建多种风险预测模型。

5.2.3　实验设置

针对多维安卓特征指标（特征集）的计算，我们分别使用 SonarQube 工具收集 Java 代码指标 (详见表4.1)，使用工具 aDoctor 收集 15 种 Android 代码异味（详见表4.2），使用 DACS 工具收集自定义的 15 种 Android 代码异味（详见4.2.1）。针对初始安全风险等级（标签）的计算，我们首先使用 Androrisk 工具得到初始风险分析结果。基于该结果，我们邀请了某大学的三名硕士研究生完成人工修

正工作（评判标准见4.3.3节），以便得到相对正确的风险标签结果。所有的实验都是在一个 64 位的 Ubuntu 系统上进行的，该系统有 32gb 的 RAM 和 2.5ghz 的 intelxeon CPU。

我们首先对特征集采取了预处理，包括特征标准化、独立化、类不平衡处理，并应用不同的机器学习方法建立了几种风险预测模型。具体来说，我们使 Sklearn 库中基于 Python 语言的开放源码包，实现了特征预处理操作，即，（1）*sklearn.preprocessing* 模块中的 *scale* 包实现 z-score 标准化（2）*sklearn.decomposit-ion* 模块中的 *PCA* 包实现 PCA 降维（3）*imblearn.over_sampling* 模块中的 $SMOTE$ 包实现不平衡处理。此外，我们分别采用九种机器学习算法构建分类模型，即，（1）*sklearn.naive_bayes* 模块中的 *GaussianNB* 包实现朴素贝叶斯（2）*sklearn.neighbors* 模块中的 *KNeighborsClassifier* 包实现最近邻（3）*sklearn.linearmodel* 模块中的 *LogisticRegression* 包实现逻辑回归（4）*sklearn.ensemble* 模块中的 *RandomForest-Classifier* 包实现随机森林（5）*sklearn.tree* 模块中的 *DecisionTreeClassifier* 包实现决策树（6）*sklearn.svm* 模块中的 SVC 包实现支持向量机（7）*sklearn.ensemble* 模块中的 *GradientBoostingRegressor* 包实现梯度提升树（8）*sklearn.neural_netwo-rk* 模块中的 *MLPClassifier* 包实现多层感知器（9）*tensorflow* 库实现卷积神经网络。

为了降低模型训练结果的不确定性，我们采用了 10 倍交叉验证法，并将该过程循环进行了 10 次。在每个 10 倍交叉验证过程中，我们又随机地将应用程序集按照 9:1 的比例划分出两份数据，分别用于模型的训练和模型的测试。在重复 10 次验证后，我们将结果进行汇总，并将其平均结果作为最终结果来评估预测模型的性能。预测模型的构建与分析均是由 Python 语言实现的，其中有关机安卓特征的提取，特征数据的预处理和机器学习分类模型的训练和测试依均赖于开源 scikit-learn 库和 tensorflow 库。

5.2.4　评价指标

（1）为了回答问题一，我们需要某个相关系数来研究多维安卓特征指标（包括 Java 静态代码指标和 Android 代码异味）之间的依赖性。目前衡量指标间相关性的经典系数有两个：Spearman 相关系数和 Pearson 相关系数。其中，Pearson 相关系数有两个假设条件：（1）两两比较的数据是从正态分布的数据集中得到的（2）数据间具有相等单位，并且其零点是相对的不是绝对的。对于这两个先决条件，我们需要测量的特征指标均不满足。因此我们采用 Spearman 相关系数计算特征指标之间的相关性。Spearman 相关系数通常被用于计算排序后的变量之间的相关性。我们需要把 n 个初始数据被变更成定序数据。假设 x_i 和 y_i 是一

对呈单调递增趋势的定序数据，则 Spearman 相关系数计算公式为5.6。

$$\rho = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}} \tag{5.6}$$

在计算了 Spearman 的相关系数之后，我们进一步解释了相关系数的数值意义。根据 [116] 文献，表5.5给出了相关系数及其相应的相关水平。当系数等于 0 时，则表示两种指标之间完全无关，当系数等于 1 时，则表示两者指标之间完全相关。具体来说，我们基于 Python 语言，使用开源 *scipy.stats* 库中的 *spearmanr* 包实现了斯皮尔曼相关系数的计算。

表 5.5: Spearman 的秩相关系数

相关系数	相关水平
0.0 - 0.1	无相关性
0.1 - 0.3	弱相关性
0.3 - 0.5	中度相关性
0.5 - 0.7	中高度相关性
0.7 - 0.9	高度相关性
0.9 - 1.0	完全相关

（2）为了回答问题二，我们使用准确度、精度、召回率和 F1 分数四个指标来评估不同预测模型的有效性。它们的定义如下。

- **准确度**是一个直观的性能指标，计算了数据的平均值和真实值之间的差异。公式如下。

$$A(M) = \frac{TN + TP}{TN + FP + FN + TP} \tag{5.7}$$

- **精度**精度是指正确标记为正例的样本数与所有标记为正例的样本数之比。公式如下。

$$P(M) = \frac{TP}{TP + FP} \tag{5.8}$$

- **召回率**是覆盖率的一个指标，表示在实际为正例的样本中被分为正例的比率。公式如下。

$$R(M) = \frac{TP}{TP + FN} \tag{5.9}$$

- **F1 分数**同时兼顾了准确度和召回率。它是两者间的调和平均值。当精度和召回率之间有某种平衡关系时，F1 分数会变高。公式如下。

$$F1 = \frac{2TP}{2TP + FP + FN} \tag{5.10}$$

在本文中，我们对安卓应用程序的安全风险等级进行预测。对于某类安全风险等级 x，我们的输出等级为 x 和其它等级 y。任何的安卓应用程序只能得到 4 种输出结果。其中，TN（真阴性）表示预测风险等级为 x，实际风险等级也是 x。TP（真阳性）表示预测风险等级为 y，实际风险等级也是 y。FN（假阴性）表示预测风险等级为 y，实际风险等级为 x。FP（假阳性）表示预测风险等级为 x，实际风险等级为 y。对于本文中的三类安全风险等级，我们需要针对每一种安全风险等级求出相应评价指标，然后再对这些指标求平均值。

在多分类模型中，为了评估分类模型的整体性能，我们需要计算评估指标的平均值。目前主要有两种方式来计算平均值：macro 和 weighted。假设某评估指标在二分类时的计算结果为 P。当以 macro 方式计算时，它不考虑类别的数量，计算公式为：各类别的 P 值的累加值 / 类别数量。当以 weighted 方式计算时，其计算公式为：各类别的 P 值 × 该类别的样本数量 / 样本总数量。对于类不平衡的分类模型，采用 macro 方式会有较大的偏差，采用 weighted 的方式则能更好的反应模型的好坏。因此，本文采用的是 weighted 方式计算评估指标的平均值。具体来说，我们调用开源 Sklearn 库中的函数 *accuracy*$_s$*core*()，*precision_score*()，*recall_score*() 和 $f1$*_score*()，并通过设置函数参数 *average= "weighted"*，分别计算模型的 *accuracy*，*precision*，*recall* 和 $F1$ score。

除了以上的经典评估指标外，Czibula 等人 [117] 认为 ROC（Receiver Operating Characteristic）是反映分类精度的最佳指标。为了更直观地从九种模型中选出最佳预测模型，我们也绘制出 ROC 曲线来评估预测模型的性能。ROC 是一个很直观的指标来揭示分类器的准确性。如第3.3节所述，分类模型中的训练集通常是类间不平衡的，即高风险应用的数量远远少于低风险应用的数量。因此训练集的数据分布容易发生改变，造成预测结果的随机性。而 ROC 曲线能忽略这种差异，保持一致性。AUC 指标计算了 ROC 曲线下方的面积大小，其值介于 0 和 1 之间。与其他性能评估指标相比，它的方差更低。AUC 是一个数值型指标，它可以定性评价分类器的分类性能。该数值越高，模型的分类效果越好。在 ROC 曲线中，x 轴表示了 FPR（假阳性率），y 轴表示了 TPR（真阳性率），它描述了 TP（真阳性）和 FP（假阳性）之间的关系。TPR 和 FPR 的定义如下。

- **真阳性率 (TPR)** 表示正确识别的阳性样本的比例。其计算方法与公式5.9中计算召回率的方法相同。

- **假阳性率 (FPR)** 表示被错误识别为阳性样本的阴性样本的比例。公式如下。

$$FPR = \frac{FP}{FP + TN} \tag{5.11}$$

具体来说，我们使用 *sklearn* 库中的 *metrics.roc_curve* 函数绘制出每个预测模型的 ROC 曲线图。然而需要注意的是，前文所总结的 ROC 曲线是针对二分类模型的而言的。在本文中，我们构建的是一个多分类（三种安全风险等级）的预测模型，对于 ROC 曲线的绘制方式有两种。也就是，*sklearn.metrics.roc_auc_score* 函数中的参数 *average* 有两种选项：为 *'macro'* 和 *'micro'*。（1）当 *average = macro* 时，对于每一种类别，都可以单独绘制出一条 ROC 曲线。当有 n 类时，对应有 n 条 ROC 曲线。然后根据这 n 条 ROC 曲线的平均值，画出最终的 ROC 曲线。（2）当 *average = micro* 时，我们先把多分类问题转化成一个二分类问题。转换完成后，对于每一个样本，它的标签只有两种可能：0 和 1。1 表明了它的实际类别，0 就表示其他的类别。所以，在转换为二分类问题后，我们可以根据上述公式直接绘制 ROC 曲线。

（3）为了回答问题三，我们利用梯度提升树算法，计算出每一个特征指标在预测模型中的重要性。我们使用每个特征的基尼不纯度来衡量特征指标在模型中的重要度。基尼不纯度是一种用于分裂决策树的方法，它可以选择最优的节点分割方案以构建决策树。通俗来讲，基尼不纯度度量了数据集中的任意元素被随机标记时，被错误标记的概率。它衡量了某集合的无序程度，其计算公式为5.12。

$$I_G(f) = \sum_{i=1}^{m} f_i(1 - f_i) = \sum_{i=1}^{m} f_i - \sum_{i=1}^{m} f_i^2 = 1 - \sum_{i=1}^{m} f_i^2 \tag{5.12}$$

其中，f_i 表示某随机事件发生的概率。基尼不纯度是一个介于 0 到 0.5 之间的数字。这个数值越小，意味着基尼纯度越高。它代表数据集越有序，模型的分类效果越好。考虑极致情况，当基尼不纯度等于 0 时，表示所有预测结果都与真实类别保持一致。当 $f_1 = f_2 = \ldots = f_m = \frac{1}{m}$ 时，基尼不纯度最高，纯度最低，集合越混乱。具体来说，我们基于 Python 语言，使用 *collections* 库中的 *Counter* 包和 *operator* 包实现了基尼不纯度的计算。

5.2.5　实验结果分析

5.2.5.1　问题一的结果分析

我们使用了 Spearman 的秩相关系数来研究混合指标（即本文中的 Java 静态代码指标和 Android 代码异味）之间的相关性。这种系数对数据的先决条件不

严格，不考虑数据的总体分布和样本大小。图5.1显示了 Spearman 相关系数的热图。从图5.1 中我们可以发现，Java 静态代码指标之间的 Spearman 相关系数大小在 0.8 到 1 之间，表明了这些指标存在高度的相关性（根据表5.5)。这意味着在建立预测模型时，本文使用的一些 Java 静态代码指标在很大程度上是冗余的，存在多重共线性，这会导致几个问题：（1）增加不必要的特征维度（2）容易造成模型的过拟合（3）加大模型的学习难度，增加时间和空间的复杂度（4）特征间相互影响，影响模型的训练结果。这些也是我们在建模时对这些特征指标进行 PCA 预处理的几个主要原因。

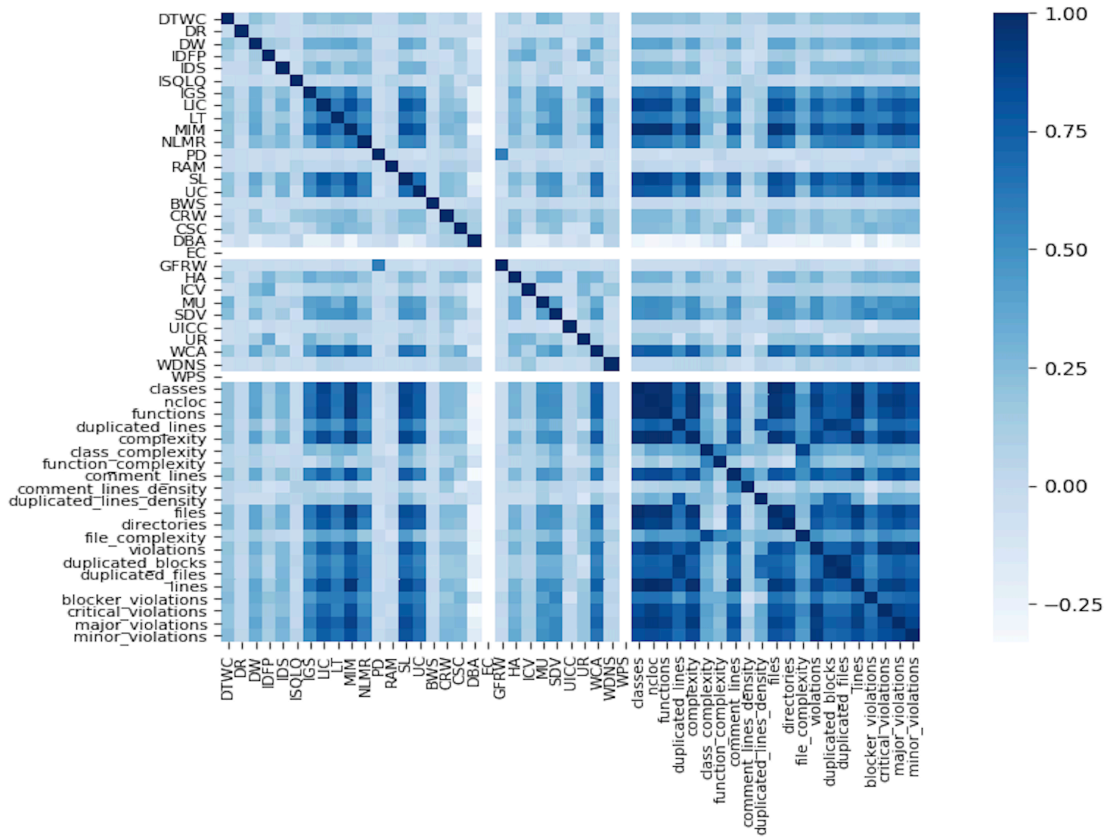


图 5.1: 混合指标（即 Java 静态代码指标和 Android 代码异味）之间的相关性

我们从图 5.1中可以看到，Android 代码异味之间的相关系数在 0.0 到 0.2 之间，显示出从无到弱的相关性（根据表5.5)。此外，Java 静态代码指标和 Android 代码异味之间的相关系数在 0.4 左右，这意味着它们之间的相关性也是比较小的。因为 Android 代码异味与其他特征指标之间存在的相关性非常低，这表明了代码异味的引入会为预测模型带来有效的，有意义的信息，在更好地解释模型的同时，也不会给模型带来过拟合问题。这让我们更加确定，通过将 Android 代码异味和 Java 静态代码指标组合，可以更准确地预测出 Android 应用程序的安

全风险等级。

很多论文 [15, 16] 都表明了特征指标和代码大小有比较强烈的关系。图5.1也展示了一些表示代码大小的指标（如，代码行数（lines），文件数（files）和类数（classes））和 Android 应用程序中代码异味的相关性。我们发现，大多数的代码异味和代码大小指标之间的相关系数在 0.0 到 0.2 之间，这表明它们之间的相关性不是特别强烈，然而，有一些代码异味（即，内部 Getter 和 Setter（IGS）、内部类泄露（LIC）、线程泄露（LT）和非静态内部方法（MIM））和代码大小有高度关联性（斯皮尔曼系数在 0.6 左右）。

问题一的回答：（1）Android 代码异味之间是无相关性的（2）Android 代码异味和 Java 静态代码指标之间是呈弱相关性的（3）大多数 Android 代码异味和代码大小无关，少数异味（内部 getter 和 setter（IGS）、内部类泄露（LIC）、线程泄露（LT）和非静态内部方法（MIM））和代码大小呈高度相关性（4）Java 静态代码指标之间是呈高度相关性的。因此我们认为，Android 代码异味的引入能更好地解释、训练模型。

5.2.5.2　问题二的结果分析

为了回答问题二，我们将九种机器学习算法（如第4.4.1节所述）应用于一个包含 4575 个 Android 应用程序的数据集，构建了一系列的风险预测模型。为了更好地研究 Java 静态代码指标，特别是 Android 代码异味在预测应用程序安全风险等级方面的作用，我们分别以三个不同维度的安卓指标作为模型特征，应用机器学习算法，建立应用程序的风险预测模型。这三种特征集为（1）只有 Android 代码异味，（2）只有 Java 静态代码指标及（2）前两种指标的组合。通过评估这三种特征集所建立的预测模型，我们可以探索代码异味的引入，是否真的能够有效的提高预测模型的准确率。

如第4.3.4节所示，我们需要对安卓特征集先进行预处理操作。为了防止此类操作会对模型的预测结果造成影响，从而不能有力判断代码异味在模型预测中的影响力。我们排列组合了各种数据处理手段，即，（1）使用 / 不使用了 PCA 算法对所有特征指标进行降维处理。（2）使用 / 不使用 SMOTE + ENN 算法进行类不平衡处理，旨在将一些影响模型的因素排除，从而判断代码异味是否真的能提高模型的预测能力。在表5.6中，我们完整展示了各种训练环境下的模型评估结果。

表 5.6: 基于 Java 静态代码指标和 Android 代码异味构建的安全风险等级预测模型

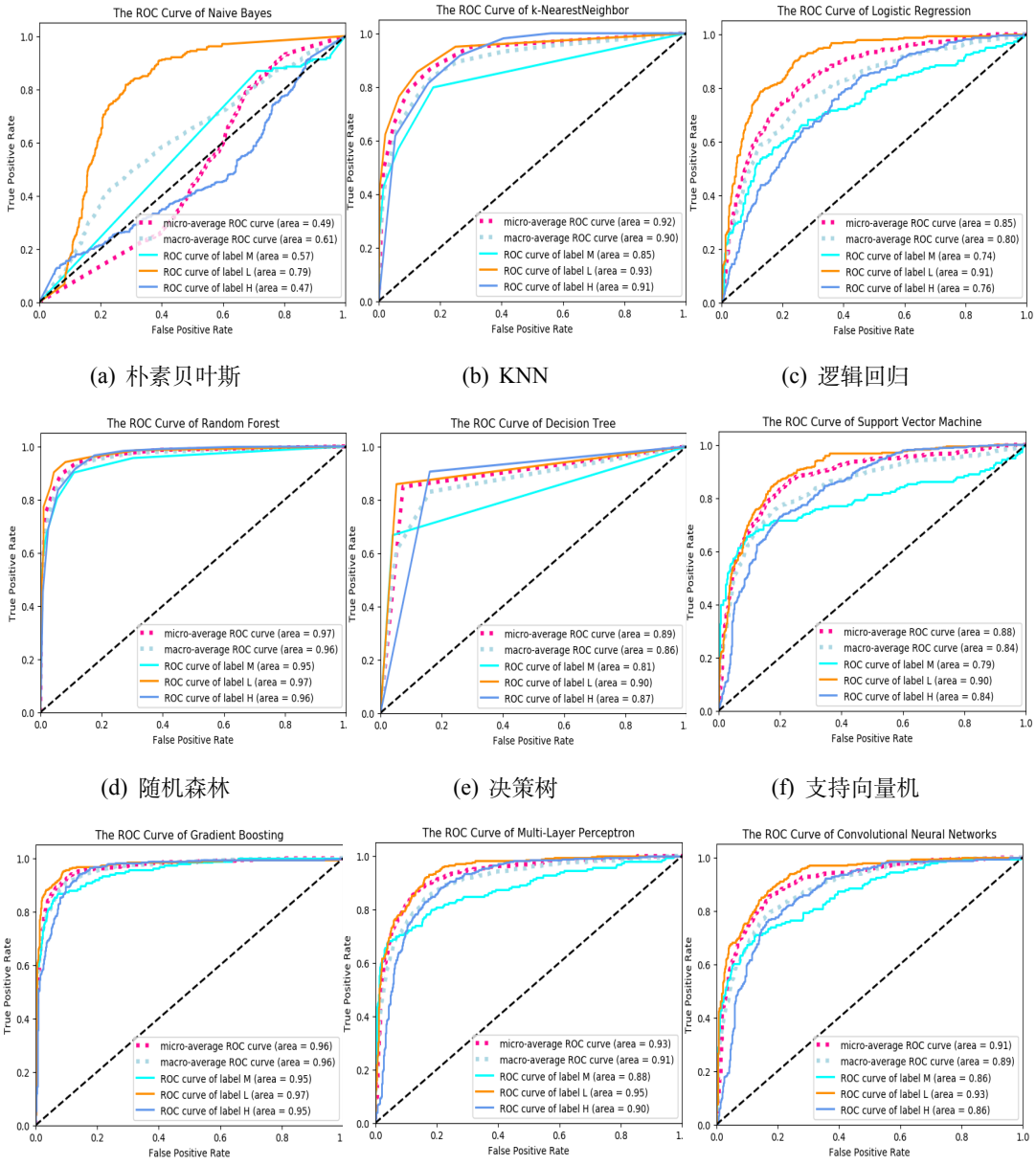
分类器	PCA	SMOTEENN	指标							
			Java 静态代码指标				Android 代码异味 / 混合指标			
			精度	召回率	F1 分数	准确度	精度	召回率	F1 分数	准确度
Naive Bayes	无	无	0.518	0.558	0.515	0.558	0.395 / 0.521	0.433 / 0.561	0.403 / 0.513	0.432 / 0.561
		有	0.542	0.262	0.218	0.262	0.389 / 0.536	0.460 / 0.261	0.380 / 0.221	0.462 / 0.263
	有	无	-	-	-	-	-	-	-	-
		有	-	-	-	-	-	-	-	-
KNN	无	无	0.598	0.623	0.600	0.623	0.571 / 0.601	0.606 / 0.631	0.577 / 0.601	0.606 / 0.631
		有	0.801	0.798	0.791	0.797	0.723 / 0.802	0.726 / 0.802	0.720 / 0.796	0.726 / 0.802
	有	无	0.582	0.621	0.593	0.621	0.578 / 0.612	0.613 / 0.635	0.583 / 0.608	0.613 / 0.635
		有	0.782	0.778	0.771	0.778	0.701 / 0.801	0.703 / 0.792	0.701 / 0.785	0.703 / 0.792
Logistic Regression	无	无	0.581	0.631	0.578	0.631	0.533 / 0.581	0.585 / 0.631	0.466 / 0.583	0.585 / 0.631
		有	0.681	0.678	0.658	0.678	0.660 / 0.690	0.641 / 0.686	0.605 / 0.668	0.641 / 0.686
	有	无	0.572	0.628	0.565	0.628	0.432 / 0.570	0.575 / 0.622	0.425 / 0.561	0.575 / 0.622
		有	0.650	0.635	0.612	0.635	0.664 / 0.660	0.640 / 0.648	0.600 / 0.626	0.640 / 0.648
Random Forest	无	无	0.646	0.661	0.638	0.661	0.603 / 0.658	0.623 / 0.668	0.606 / 0.642	0.623 / 0.668
		有	0.881	0.882	0.880	0.882	0.760 / 0.885	0.760 / 0.885	0.760 / 0.885	0.760 / 0.885
	有	无	0.655	0.665	0.640	0.665	0.608 / 0.635	0.622 / 0.652	0.612 / 0.628	0.622 / 0.652
		有	0.840	0.840	0.838	0.841	0.735 / 0.834	0.735 / 0.835	0.735 / 0.834	0.735 / 0.835
Decision Tree	无	无	0.602	0.600	0.601	0.598	0.570 / 0.603	0.570 / 0.600	0.569 / 0.602	0.570 / 0.600
		有	0.835	0.836	0.835	0.836	0.733 / 0.836	0.732 / 0.837	0.731 / 0.835	0.732 / 0.837
	有	无	0.591	0.583	0.586	0.583	0.591 / 0.593	0.585 / 0.594	0.586 / 0.593	0.585 / 0.594
		有	0.782	0.785	0.783	0.785	0.701 / 0.801	0.698 / 0.802	0.698 / 0.800	0.698 / 0.802
SVM	无	无	0.575	0.640	0.602	0.640	0.629 / 0.617	0.583 / 0.642	0.447 / 0.603	0.583 / 0.642
		有	0.720	0.701	0.686	0.701	0.673 / 0.737	0.635 / 0.715	0.606 / 0.701	0.635 / 0.715
	有	无	0.598	0.661	0.627	0.661	0.631 / 0.691	0.590 / 0.660	0.465 / 0.625	0.590 / 0.660
		有	0.735	0.718	0.709	0.718	0.679 / 0.749	0.645 / 0.727	0.616 / 0.717	0.645 / 0.727
Gradient Boosting	无	无	0.662	0.678	0.652	0.678	0.615 / 0.664	0.633 / 0.680	0.611 / 0.655	0.633 / 0.680
		有	0.889	0.888	0.885	0.888	0.751 / 0.893	0.750 / 0.893	0.749 / 0.891	0.751 / 0.893
	有	无	0.646	0.665	0.638	0.665	0.616 / 0.639	0.633 / 0.662	0.610 / 0.633	0.633 / 0.662
		有	0.811	0.808	0.805	0.809	0.723 / 0.837	0.720 / 0.833	0.712/ 0.829	0.720 / 0.833
MLP	无	无	0.615	0.653	0.622	0.653	0.596 / 0.641	0.628 / 0.648	0.598 / 0.622	0.628 / 0.648
		有	0.768	0.765	0.760	0.765	0.683 / 0.796	0.672 / 0.793	0.653 / 0.788	0.672 / 0.793
	有	无	0.676	0.670	0.635	0.670	0.575 / 0.626	0.630 / 0.662	0.593 / 0.632	0.630 / 0.662
		有	0.725	0.723	0.716	0.723	0.674 / 0.769	0.661 / 0.763	0.638 / 0.755	0.661 / 0.763
CNN	无	无	0.590	0.650	0.616	0.650	0.566 / 0.615	0.625 / 0.648	0.593 / 0.616	0.625 / 0.648
		有	0.670	0.696	0.685	0.696	0.673 / 0.767	0.640 / 0.761	0.599 / 0.755	0.640 / 0.761
	有	无	0.580	0.631	0.563	0.631	0.566 / 0.588	0.633 / 0.647	0.591 / 0.605	0.633 / 0.647
		有	0.682	0.665	0.645	0.665	0.675 / 0.751	0.645 / 0.742	0.611 / 0.735	0.645 / 0.742

通过分析表格中的数据，我们可以发现，当我们单独使用 Java 静态代码指标，和单独使用 Android 代码异味作为安卓特征构建预测模型时，它们的模型评价指标（精度、召回率、F1 分数和准确度）的值差异不大，这意味着 Android 代码异味和 Java 静态代码指标对于安全风险等级预测模型有同样重要的影响力。此外，如表5.6所示，我们没有评估 naive bayes 算法构建的预测模型（某些评价

指标为空白），这是因为该算法要求特征指标的值是非负数的，在对特征指标采取 PCA 降维后，其重新构建的特征值有的为负数，因此，无法基于 naive bayes 算法 + PCA 算法构建相应的风险预测模型。与单类的特征指标相比，将 Java 静态代码指标和 Android 代码异味作为组合指标，并应用机器学习方法，可以构建出预测性能更好的模型。其中，精度、召回率、F1 分数和准确度的平均值分别为 0.70、0.75、0.70 和 0.80。这表明 Java 静态代码指标和 Android 代码异味在预测应用程序的安全风险等级方面是相辅相成的，代码异味的引入确实能提高模型的预测性能。

如第4.3.5节所述，我们基于九种流行的机器学习算法建立了不同的风险预测模型。为了更好地探索不同的学习算法的训练能力，从这些模型中找出性能最佳的预测模型。针对每一个风险预测模型，我们绘制出相应的 ROC 曲线，并计算了该曲线下方的面积（AUC，大小介于 0.1 和 1.0 之间。AUC 值越大，表示当前预测模型的分类能力越好）。由于我们定义了三种不同的安全风险等级：低（L）、中（M）和高（H）。对于每一种等级，我们都绘制出相应的 ROC 曲线，同时，我们根据5.2.4所述的两种方式：macro 和 weighted，绘制出多分类问题的 ROC 曲线。如图5.2所示，图中展示了各个安全风险等级对应的 ROC 曲线以及模型的平均 ROC 曲线。另外，ROC 曲线下的面积大小（AUC）也根据 ROC 曲线进行了单独计算。

从图5.2上我们可以观察到，我们所提出的技术能更好地分类出处于高风险和低风险等级的 Android 应用程序，对这两种类别的数据有更高的预测能力。经过调查发现，处于这两个安全风险等级的应用程序的数量相对较多。对于模型的训练数据集而言，处于中风险的数据量较小，使得模型对此类数据的学习不够充分。另一个观察结果是，除了朴素贝叶斯算法外，其他算法所构建的风险预测模型的性能都很高，它们的 AUC 值的都达到了 0.9 左右。朴素贝叶斯之所以“朴素”，是因为它认为所有的特征指标都是独立的，且对模型的影响力是等同的。但是如5.2.5.1所描述，我们提取的多维安卓特征指标之间存在一定的依赖性，且对于模型的作用大小是不等的。因此，朴素贝叶斯算法在所有的学习算法中表现最不佳。此外，与其他的学习算法相比，RF（随机森林）和 GBDT（梯度提升决策树）具有更好的预测性能--RF 和 GBDT 的 AUC 的值分别为 0.97 和 0.96（本文基于 Python 语言，使用开源 scikit-learn 库，并设置默认配置训练了 RF 和 GBDT 模型）。这样的结果也符合规律。因为 GBDT 和 RF 都是经典的集成学习算法，它们在训练时生成多个决策树。与单一决策树相比，这种集成方法可以大大减少由单一决策树引起的过拟合问题 [118–120]，从而进一步提高了模型的预测性能。



(a) 朴素贝叶斯　(b) KNN　(c) 逻辑回归

(d) 随机森林　(e) 决策树　(f) 支持向量机

(g) 梯度回归　(h) 多层感知器　(i) 卷积神经网络

图 5.2: 九种风险预测模型的 ROC 曲线

问题二的回答：（1）与 Java 静态代码指标相比，Android 代码异味可以构建出更具竞争力的安全风险等级预测模型（2）它们都能被用于预测 Android 应用程序的安全风险等级，且组合效果更佳（结合这两种特征指标，我们可以得到精度为 0.89，召回率为 0.89，准确度为 0.88，F1 分数为 0.89 的风险预测模型）。（3）在基于九种机器学习算法所构建的风险预测模型中，朴素

贝叶斯算法的表现最差（AUC 的值在 0.5 左右），集成学习算法中的梯度提升决策树算法和随机森林算法表现最好（AUC 的值可以达到 0.97），优于其他的分类学习算法。

5.2.5.3　问题三的结果分析

问题三的目的是研究不同代码指标，特别是代码异味在安全风险预测中的重要性。在本文中，我们基于梯度提升树算法来识别出风险预测模型中的重要特征。对于一棵梯度提升树，我们计算了每个特征的基尼不纯度[2]，并进一步用此表示特征的重要性。图5.3展示了 Java 静态代码指标，以及 Android 代码异味在安全风险等级预测模型中的重要性。从图5.3中我们可以发现，有 45 个特征贡献了 99% 的累积重要性，而 6 个没有贡献的特征指标分别是：Webview 域明文保存密码（WPS）、暴露的剪贴板（EC）、低效 SQL 查询（ISQLQ）、可调试版本（DR）、证书验证不正确（ICV）和不安全的随机数（UR）。在有贡献度的特征集中，我们可以发现违规代码 (violations)、行 (lines)、非静态内部方法（MIM）、弱加密算法（WCA）和 SD 卡访问（SDV）等指标对预测模型的性能有较大的影响。这意味着开发人员在应用程序开发过程中，应该更加注意避免这些代码异味，以防造成应用程序的高风险。例如，如果不访问类的任何内部属性的方法不是静态的（static）[22]，那么就引入 MIM 异味。为了避免 MIM 带来的潜在安全风险，开发人员最好将此类非静态方法设置为静态方法。

此外，图5.3中的结果还说明，代码大小和应用程序风险之间有很强的关联性。一个应用程序越复杂，代码量越大，越容易带来应用程序的安全漏洞。在安卓代码异味中，不良的违规代码 (bad violations) 异味对 Android 应用程序的风险有着重要影响，而一些代码异味对应用程序风险的影响较小。在我们查看数据集后发现，在 4575 个应用程序中，只有 65 个应用程序具有 WPS 异味，24 个应用程序具有 EC 异味，7 个应用程序具有 DR 异味，3 个应用程序具有 ICV 异味，1 个应用程序具有 UR 异味，9 个应用程序具有 ISQLQ 异味），这导致了模型的训练数据集不足，学习不充分不完整，造成此类代码异味的对模型的贡献度低。此外，在分析代码异味时我们发现，在我们的研究中，一些与 Android 安全性自然相关的代码异味（例如公共数据（PD））只显示出与应用程序风险的弱相关性，在预测模型中的重要性较低。其主要原因也是因为在我们的数据集中，只有少数应用程序包含此类代码异味。因此，在未来的工作中，（1）我们还需要

[2]https://en.wikipedia.org/wiki/Decision_tree_learning#Gini_impurity

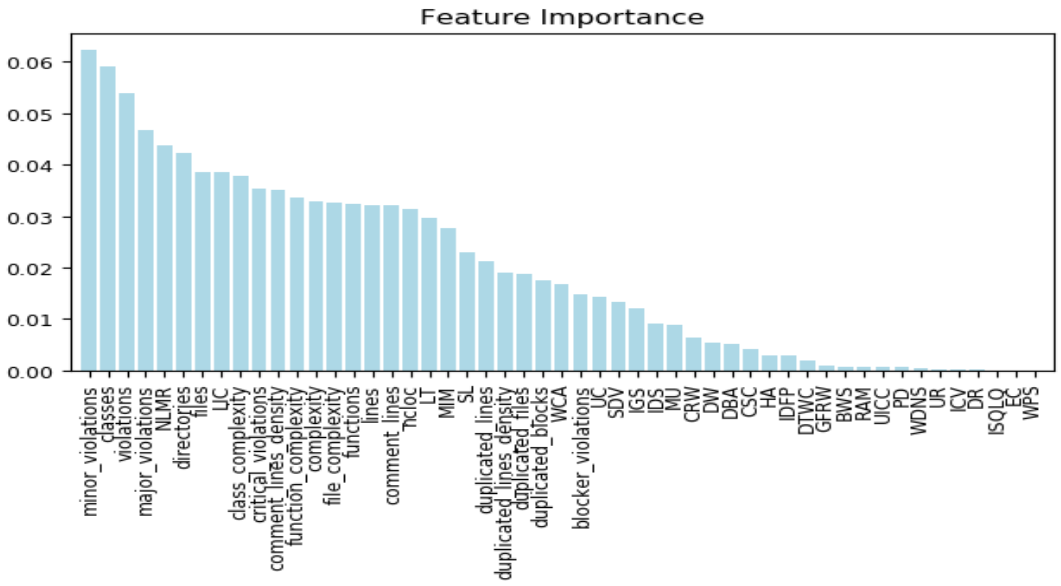


图 5.3: Java 静态指标和 Android 代码异味指标的重要性

收集更多的安卓应用程序作为实验数据，尽量做到代码异味的全覆盖，以便更好地对每个代码异味进行研究。（2）定义更多更新的安卓代码异味，总结出大多数安卓应用程序所包含的不良的代码结构。总体来说，与 Java 静态代码指标相比，虽然 Android 代码异味的重要性不是最高的，但是代码异味的重要性的绝对值只是略小于 Java 代码静态指标的值。这表明了这两种特征指标在帮助预测 Android 应用程序的安全风险等级方面都有一定的贡献度。

问题三的回答：（1）在构建安卓应用程序的风险预测模型时，Java 静态代码指标和 Android 代码异味都有一定的贡献度。（2）对于代码异味而言，非静态内部方法（MIM）和弱加密算法（WCA）在风险预测模型中的重要性较高，贡献度较大，这说明这些代码异味容易造成应用程序的高风险，开发者在开发应用程序时应注意修复这些代码异味。

5.3　实验三：故障数计数模型的建立与评估

如第4.4节所描述，为了评估安卓应用程序的质量风险，本文建立了故障数模型，其中故障数统计了应用中有关安全，内存，性能等各项问题和故障数量。具体来说，本文以故障关键字为出发点，在 GitHub 网站上检索出包含此类关键字的开源项目，并在一个版本周期内，统计出应用程序的故障数。由于原始数据

中的故障数分布不服从正态分布,且离散程度较大，因此我们分别选取了负二项回归算法和零膨胀负二项回归算法作为故障数拟合函数。接着我们选取了实验二中的 5 种自定义代码异味，基于这些代码异味和 NCLOC 指标（代码行数，非空白行/非注释行且包含字符的物理行数）拟合出应用程序故障数的变化。

5.3.1　实验目的

基于实验二的结果,我们发现了弱加密算法（WCA）代码异味,头文件（HA）代码异味，数据备份（DBA）代码异味，恶意压缩（MU）代码异味和 SD 卡访问（SDV）代码异味会对安卓应用程序造成较大的安全隐患，为了进一步研究这五种代码异味对应用程序质量的影响，以及这些影响的具体大小。我们设计了实验三用于研究这五种代码异味和应用程序故障数之间的关系，其中故障数统计了安全、性能、内存等各项故障数量。我们讨论了两个研究问题如下所示：

问题一: 哪些代码异味和应用程序故障数呈正相关，哪些代码异味和故障数呈负相关，哪些代码异味和故障数无关？

问题二: 哪一种回归算法（负二项回归、零膨胀负二项回归）构建的故障数模型拟合效果更好？

问题一的回答可以研究五个代码异味和故障之间是否存在不一致的关系。在开发过程中，有哪些代码异味会对应用程序的质量（包括安全，性能，内存等）造成影响，哪些代码异味不会影响应用程序的质量。在代码异味确实影响应用程序的质量（造成应用程序故障）的地方，这种影响的相对大小是多少。帮助从业者和研究人员更好地将研究集中在对故障有较大影响的代码异味上。

问题二的回答可以研究在本次实验的数据集上，哪一种回归算法的训练效果更好，帮助研究者在以后研究此类问题时，更好地进行算法选择。

5.3.2　实验数据

如4.4.2节所示，我们在 645 个开源安卓应用程序上进行了实验，应用了负二项回归算法建立了计数模型。基于实验二中所提取的 30 种安卓代码异味，我们从中选择了五种异味（弱加密算法（WCA）异味，头文件（HA）异味，数据备份（DBA）异味，恶意压缩（MU）异味和 SD 卡访问（SDV）异味），因为它们最经常出现在代码中，且对应用程序的安全和隐私性影响较大。要自动化检测代码异味，必须对这些异味有良好的定义。由于所研究的 5 种异味是我们自定义的，我们能对这些异味有最精确和最完善的认知（详细定义见4.2.1节）。

如4.4.1节所示，我们收集了安卓应用程序在一个版本内的故障数，涵盖了安全、内存、性能等各项问题和故障。为了直观地看到代码异味和故障数之间的

变化关系，我们画出了原始数据的散点图矩阵。图5.4提供了应用程序中数据分布的进一步信息，显示了我们正在研究的安卓代码异味的分布，即，五种自定义的代码异味之间的关系。图5.4表明了代码异味的密度和分布在每个应用程序中是不同的，其中，每一张小图的横纵坐标分别为两种不同的代码异味。

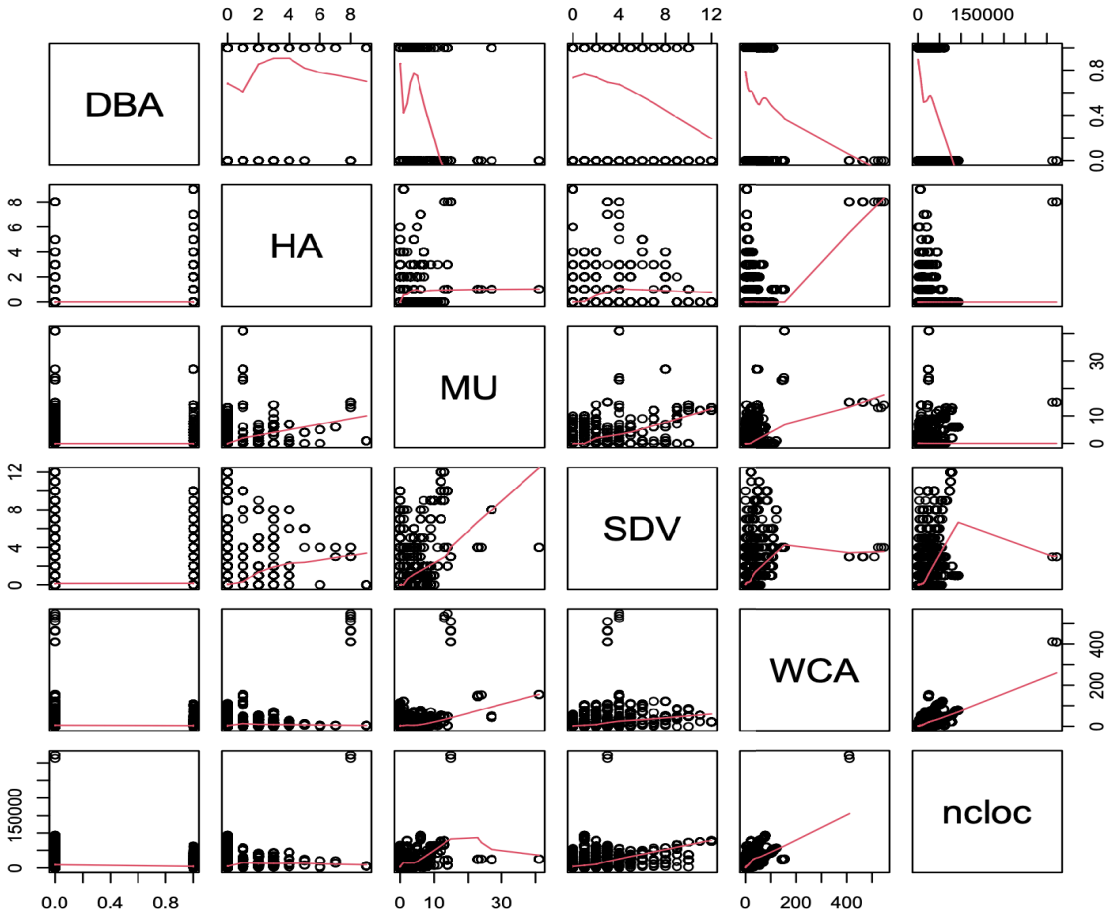


图 5.4: 代码异味矩阵散点图

此外，图5.1展示了代码异味之间的相关性。我们可以看到，除了 MU 异味和 SDV 异味之间有较强的相关性，以及 HA 异味和 WCA 异味之间有较弱的相关性外，大多数相关性都很低（<0.30）。这些代码异味是独立的且互不影响，满足了某些回归模型的先决条件之外，还避免了实验结果的随机性，保证了实验设计的合理性。在第2.4节，我们介绍了代码异味和代码大小（如 NCLOC 指标）之间的相关性，图5.1中也反映出了代码异味和代码行数之间存在一定的关联。因此，在研究代码异味和故障数的关系时，我们也一同将 NCLOC 指标纳入考虑。

我们所研究的五种异味都是基于方法级别的异味，每种异味数量都是每个应用程序文件的计数值（即，我们在方法体内检测并累加这些代码异味）。这个

整数值表示该文件中出现异味的次数累加值。我们的回归计数模型是以应用程序文件中的故障数为因变量，NCLOC 数为因变量，代码异味为自变量建立的。

5.3.3　实验假设

本研究的目的是调查代码异味和应用程序中故障数（涵盖了安全，内存，性能等各项问题数）的关系。考虑到应用程序的文件大小，本次调查研究了包含五种代码异味的文件是否比不包含这些异味的文件更容易出现故障。判断某代码异味对应用程序的故障是否有影响时，我们可以借鉴数学中的“反证法”思想，即，先假设，后拒绝。（1）假设某代码异味对应用程序的故障数量无影响，（2）对数据进行统计分析，（3）根据分析结果，发现“某代码异味对应用程序的故障数无影响”这件事的可能性太低，（4）拒绝“某代码异味对应用程序的故障数无影响”的假定，认为某代码异味会影响应用程序中的故障数。这里一开始的“某代码异味对应用程序的故障数无影响”的假设就是原假设。具体来说，我们在 $p < 0.05$ 的显著性水平上验证以下假设的正确性：

假设 1：弱加密算法（WCA）代码异味对应用程序中的故障数量没有影响（不管是单独的还是与其他指标结合的）。

假设 2：头文件（HA）代码异味对应用程序中的故障数量没有影响（不管是单独的还是与其他指标结合的）。

假设 3：数据备份（DBA）代码异味对应用程序中的故障数量没有影响（不管是单独的还是与其他指标结合的）。

假设 4：恶意压缩（MU）代码异味对应用程序中的故障数量没有影响（不管是单独的还是与其他指标结合的）。

假设 5：SD 卡访问（SDV）代码异味对应用程序中的故障数量没有影响（不管是单独的还是与其他指标结合的）。

假设 6：代码行数（NCLOC）与应用程序中的故障数量没有影响（不管是单独的还是与其他指标结合的）。

P 值是我们是否接受原假设的评判指标。在定义了原始假设前提下，P 值表示了由于随机错误导致原始假设不成立的概率。当 P 值比较小时，否定原假设的错误率就很小，所以我们必须拒绝原假设事件，当 P 值比较大时，否定原假设的错误率就很大，所以我们需要接受原假设事件。

当显著性水平设置为 0.05 时，如表5.7所示，当 p 值低于 0.05 时，我们认为由偶然因素导致的“代码异味对应用程序故障无影响“事件的概率较小，因此不能接受原假设，换言之，接受“代码异味对应用程序的故障数存在显著影响”的事实。假设 P 的值大于 0.05，则由随机误差导致的“代码异味对应用程序的

故障数无显著影响”事件的概率较高，这表明我们没有足够的证据拒绝“代码异味对应用影响的故障数无影响”的假设，保守起见接受原假设。

表 5.7: P 值的数据解释

P 值	偶然性	对原假设	统计意义
P > 0.05	偶然出现的概率大于 5%	不能否定原假设	代码异味对应用程序的故障数无显著影响
P < 0.05	偶然出现的概率小于 5%	可以否定原假设	代码异味对应用程序的故障数有显著影响
P < 0.01	偶然出现的概率小于 1%	可以否定原假设	代码异味对应用程序的故障数有非常显著影响

5.3.4　实验设置

我们的目标是在模型拟合程度高的前提下，模型中的参数越少越好。旨在构建简单而高拟合的故障数回归模型。本文以此为目标，分别应用了负二项回归算法和零膨胀负二项回归算法，建立了安卓应用程序中的故障数与代码异味间的关系模型, 通过反复的模型评估和模型选择, 最终确定了模型的自由参数项（代码异味）。对于每个数据集，我们反复地对数据进行拟合，旨在构建一个与数据最匹配的交互模型。具体来说，我们构建故障数模型的步骤如下所示。

（1）一阶模型中包含所有的独立自变量, 即五种自定义的代码异味, NCLOC 指标以及它们的组合变量。它将每个自变量对故障的影响从所有其他自变量的影响中分离出来，并且分析不同的自变量对故障的影响。

（2）我们从一阶模型中选择与故障数显著相关的项（P 值小于 0.05 的项），建立一个更简单的数据模型。

（3）重复第二步，继续选择与故障数显著相关的项去构建更简单和更适合的模型，直到所有的项均与故障数显著相关。

故障数的回归模型的构建和评估均是由 R 语言实现的。我们使用 MASS 包中的 glm.nb 函数拟合负二项回归函数，使用 pscl 软件包中的 zeroinfl 函数实现了零膨胀负二项回归模型，并评估这两个回归模型对数据的拟合程度。

5.3.5　评价指标

为了比较负二项回归模型和零膨胀回归模型对安卓应用程序的故障数与代码异味的关系的拟合程度，本文计算了故障数模型的 AIC 值和 BIC 值，以选择

拟合度最高的计数模型。其中，AIC 信息准则是在信息熵的基础上，寻找涵盖自变量最少但拟合度最高的模型。公式如5.13所示。

$$\mathrm{AIC} = -2\ln(L) + 2\alpha \tag{5.13}$$

其中 L 表示的是似然函数，α 表示可估计的自由参数的数量。AIC 比较了回归函数拟合出的值与真实值的偏差来评估一个模型的好坏。AIC 越小，代表模型的拟合值和真实值的差异越小，即回归模型对数据的拟合度越高。

BIC 信息准则是贝叶斯统计中用来在两个或多个备选模型之间进行选择的一个评估指标。虽然 AIC 和 BIC 的模型评估标准有所差异，但是它们之间存在很多的相似之处，例如，它们对惩罚项有类似解释。如公式5.13所示，当似然值越大（模型越精确），模型中的自有参数个数越多（模型越简单）时，AIC 的值就会越小，表示模型表现更优。在数据量比较大的情况下，BIC 设置了一个比 AIC 权重更大的惩罚参数（ln(n)），用于计算出最优参数数量，缓解模型的过拟合问题。此外，BIC 还研究了模型的数据量。在数据样本过多时，该评价指标可以帮助模型在精度和复杂度之间做好平衡工作。BIC 的计算公式为如5.14所示。

$$\mathrm{BIC} = -2\ln(L) + \alpha\ln(n) \tag{5.14}$$

其中，L 表示在最大似然值下进行评估时，给定数据后，测试模型的可能性。α 表示模型估计的参数数，n 是样本大小，即观察值的数量或数据点的数量。BIC 的值越小，意味着模型的训练效果越优。

5.3.6　实验结果分析

5.3.6.1　问题一的结果分析

（1）NB 模型的拟合结果分析

为了研究代码异味对应用程序质量的影响，本文应用了 NB（负二项回归）模型对故障数（安全、内存、性能等各项问题和故障）与代码异味之间的关系进行了建模分析，表5.8展示了应用程序的一阶交互模型。表中的每一行都表示与应用程序文件中的故障数相关的代码异味。此外，代码异味的组合也通过相乘交互项进行测试，即，如果在测试文件中，如果有两种代码异味都出现其中，那么相乘交互的计算结果为 1（1×1）。例如，HA:DBA 测试了应用程序文件中出现的 HA 异味和 DBA 异味与故障数之间的关系，如果文件中同时存在这两种代码异味，则相乘交互项的计算结果为 1。同时，这些交互项会忽略文件中是否存在任何其他的代码异味。表5.8中还测试了 NCLOC 指标和故障数之间的关系。表中的截距行总结了表中未明确提及的因素对故障的平均影响。

表 5.8: 基于负二项回归算法的一阶交互故障数模型

	Estimate	**Std. Error**	**z value**	**Pr(>|z|)**
(Intercept)	-0.2615	0.0785	-2.24	**0.0138***
NCLOC	0.0013	0.0003	3.58	**0.0001*****
WCA	-0.5612	0.1843	-2.28	**0.0019****
HA	-0.3851	0.3254	-1.08	0.1995
DBA	-0.3021	0.2134	-1.72	0.1123
MU	0.0949	0.0411	2.30	**0.0116***
SDV	-0.0996	0.0384	-2.18	**0.0199***
NCLOC:WCA	0.0022	0.0013	2.67	**0.0035****
NCLOC:HA	0.0007	0.0006	-1.07	0.2810
NCLOC:DBA	-0.0022	0.0020	-1.61	0.1641
NCLOC:MU	-0.0001	0.0002	-2.48	**0.0079****
NCLOC:SDV	0.0003	0.0001	2.39	**0.0049****
WCA:HA	-0.4969	0.5088	-0.93	0.3321
WCA:DBA	0.2849	0.3899	0.81	0.4392
WCA:MU	0.0193	0.0310	0.49	0.5939
WCA:SDV	-0.0291	0.0864	-0.36	0.6895
HA:DBA	0.8248	0.4201	2.01	**0.0498***
HA:MU	0.1811	0.0568	3.09	**0.0018****
HA:SDV	-0.1722	0.1298	-1.24	0.2014
DBA:MU	0.0149	0.0309	0.52	0.6278
DBA:SDV	0.0187	0.1019	0.20	0.8455
MU:SDV	-0.0157	0.0101	-1.59	0.1017

如表5.8所示，构建的一阶交互模型在 623 个自由度（DF）上的残差平方等于 616.1，其中 df = n - k（n 样本数，k 约束条件数）。由于其残差平方不大于其 DF，因此模型符合要求。然而，表5.8显示 NCLOC，WCA，MU，SDV，NCLOC:WCA，NCLOC:MU，NCLOC:SDV，HA:DBA 和 HA:MU 是唯一与故障数量显著相关的项（p 值 < 0.05）。因此，我们只选用了这些指标构建了一个更简单的模型。

表5.9展示了一个更加简单的模型。这个模型对比一阶交互模型而言，它的拟合度更高。因为它的残差平方等于 616.4，自由度等于 635，残差平方小于自由度。此外，这个更简单的模型中 NCLOC，WCA，MU，SDV，NCLOC:WCA 和 NCLOC:MU 是与故障数更显著相关的项（p 值 <0.05）。

表 5.9: 基于负二项回归算法的二阶交互故障数模型

	Estimate	**Std. Error**	**z value**	**Pr(>|z|)**
(Intercept)	-0.2499	0.0810	-3.30	**0.0021****
NCLOC	0.0019	0.0003	4.68	**0.0001*****
WCA	-0.5838	0.2047	-2.79	**0.0039****
MU	0.0841	0.0219	3.88	**0.0001*****
SDV	-0.1219	0.0459	-2.58	**0.0082****
NCLOC:WCA	0.0019	0.0005	3.48	**0.0004****
NCLOC:MU	-0.0003	0.0001	-3.29	**0.0011****
NCLOC:SDV	0.0003	0.0002	1.41	0.1579
HA:DBA	-0.0497	0.2588	-0.23	0.8538
HA:MU	0.0272	0.0331	0.84	0.4158

我们现在使用这 6 个有效项来拟合一个更简单的模型，其结果如表5.10所示。该模型在 638 个自由度上的残差平方为 612.8（残差平方小于自由度）。这 6 个代码异味指标与应用程序的故障数之间均呈显著相关的关系（p 值 <0.05），即这几个代码指标对应用程序质量的影响最大。

表 5.10: 基于负二项回归算法的三阶交互故障数模型

	Estimate	**Std. Error**	**z value**	**Pr(>|z|)**
(Intercept)	-0.2958	0.0759	-3.92	**0.0001*****
NCLOC	0.0028	0.0003	6.31	**0.0000*****
WCA	-0.5993	0.2065	-2.99	**0.0035****
MU	0.0677	0.0198	3.31	**0.0010*****
SDV	-0.0806	0.0321	-2.66	**0.0121***
NCLOC:WCA	0.0020	0.0006	3.49	**0.0003****
NCLOC:MU	-0.0002	0.0001	-2.89	**0.0031****

（2）ZINB 模型的拟合结果分析

本文应用了 ZINB（零膨胀负二项回归）算法对故障数（安全、内存、性能等各项问题和故障）与代码异味之间的关系进行了建模，建模过程和 NB 类似。首先将所有代码异味，NCLOC 指标以及它们的交互指标作为自变量，故障数指标作为响应变量，应用零膨胀负二项回归算法构建故障数计数模型。在得到显著

相关的项之后，再基于这些项继续构建回归模型，直至模型中的所有项均显著相关。本文在此不展开列出所有模型结构，只展示最终模型结果（详见表5.11）。

表 5.11: 基于零膨胀负二项回归算法的故障数模型

	Estimate	**Std. Error**	**z value**	**Pr(>|z|)**
(Intercept)	-0.2768	0.0821	-4.15	**0.0001*** **
NCLOC	0.0031	0.0002	6.25	**0.0000*** **
WCA	-0.5980	0.2001	-2.75	**0.0025** **
MU	0.0665	0.0172	3.28	**0.0008*** **
SDV	-0.0146	0.0100	-4.83	**0.0031** **
NCLOC:WCA	0.0019	0.0005	3.38	**0.0002** **
NCLOC:MU	-0.0001	0.0001	-2.78	**0.0028** **

如表格5.10和表格5.11所示，它们的拟合结果保持一致，均显示有 6 种代码异味和应用程序故障数之间存在重要的关系。然而这种关系不总是呈正相关的，一些代码异味（弱加密算法（WCA），SD 卡访问（SDV）和 NCLOC:MU）与更少而不是更多的故障有关。而其他代码异味（代码行数（NCLOC），恶意解压缩（MU）和 NCLOC:WCA）与更多的故障相关。结果表明某些代码异味在同时出现时才会增加应用程序的故障倾向性，某些代码异味反而会减少应用程序的故障数。出现这样现象的原因不排除实验数据的不理想，由于 GitHub 上的应用程序大多不面向应用市场，它们的开发流程不规范，缺少充分的测试和维护，因此会缺失一些应用故障信息。然而，我们的实验结果仍然是有意义的。开发者需要及时重构与故障正相关的代码异味，而慎重重构那些与故障负相关的代码异味。

问题一的回答：在所研究的 5 种代码异味中，（1）弱加密算法（WCA），SD 卡访问（SDV）和 NCLOC:MU 会显著减少安卓应用程序的故障倾向性（2）代码行数（NCLOC），恶意解压缩（MU）和 NCLOC:WCA 会显著增加安卓应用程序的故障倾向性（3）头文件（HA）和数据备份（DBA）对应用程序的故障倾向性没有显著性影响。

5.3.6.2　问题二的结果分析

在本文中我们分别应用了负二项回归算法（NB）和零膨胀负二项回归算法（ZINB）构建了故障数模型，为了分析两种计数模型的优劣，即比较它们对应用

程序的故障数与代码异味关系的拟合度，我们计算了两种指标（AIC 值和 BIC 值）来评估这两种故障数模型。其中 AIC = 2k –2ln(L)，BIC = ln(n) * k - 2ln(L)。其中 L 代表了最大似然值，k 代表了模型中可估计的自由参数数，n 代表了实验数据量。本文总共选取了 5 个代码异味和 1 个 NCLOC 指标, 收集了 645 个开源安卓应用程序，因此 k = 6，n = 645。将该值代入公式后，得到相应信息准则指标。AIC 值和 BIC 值越小都代表着模型能更好地展示响应变量和自有参数之间的关系。结果表示，NB 模型的 AIC 值和 BIC 值分别为 552.68 和 557.48，ZINB 模型的 AIC 值和 BIC 值分别为 517.32 和 522.12。这说明了与 NB 模型相比，ZINB 模型对实验数据的拟合效果更好（ZINB 的 AIC 值和 BIC 值都比 NB 小），即，在我们的实验数据中，零膨胀负二项回归训练出的模型更优。

问题二的回答：在分别以负二项回归算法和零膨胀负二项回归算法所构建的两种故障数模型中，零膨胀负二项回归更适用于我们所研究的数据集（AIC=517.32，BIC=522.12）。

5.4　本章小结

本章节我们主要进行了三个实验，实验一评估了本文 DACS 工具检测代码异味的能力。在实验一中，我们引入了人工检测作为对比结果，计算其和 DACS 工具检测结果的一致性。就结果而言，我们开发的 DACS 工具可以代替人工检测工作，准确又高效的检测出 Android 应用程序中的代码异味。在实验二中，我们基于安卓代码异味和 Java 静态代码指标，应用九种流行的机器学习算法，建立了应用程序的安全风险等级预测模型。结果发现，在引入安卓代码异味后，我们可以建立更高准确率的预测模型，帮助开发者理解和识别包含异味的代码结构，进而加速异味的修复。同时，我们还绘制出 ROC 曲线，找到九种预测模型中性能最佳的模型。基于实验二中的结果，我们选择其中的五种代码异味（弱加密算法（WCA）异味，头文件（HA）异味，数据备份（DBA）异味，恶意压缩（MU）异味，SD 卡访问（SDV）异味）以及代码行数（NCLOC）指标为自变量，以应用程序中的故障数为因变量，在实验三中建立了故障数模型，进而研究这些指标对应用安全、内存、性能等全方面的影响。结果表明 NCLOC，MU 和 NCLOC:WCA 代码异味会显著增加应用程序的故障倾向性，开发者和研究人员需要对此类代码异味及时修复和重构。此外，零膨胀负二项回归（ZINB）模型对应用程序中故障数与代码异味之间的关系拟合度更高。研究人员未来在做类似研究时，可以优先考虑该模型。

第六章　总结与展望

6.1　结果有效性分析

与任何学术研究一样，在本文中也存在一些外在/内在的因素会影响研究结果的有效性。我们从三个角度简要地讨论了它们。

6.1.1　建构有效性

为了利用 Android 代码异味来帮助模型预测应用程序的安全风险等级，我们应该首先定义什么是 Android 代码异味，以及要使用什么样的代码异味。由于不同的实践者对 Android 代码异味的定义可能有不同的看法，本文检索的代码异味仅代表了少数人的意见（自定义的或 [22] 文中定义的）。同时，我们构建分类模型对 Android 应用程序的安全风险等级进行预测，分类模型只能识别出应用程序是否处于高风险状态，而没有具体定位高风险代码块。

6.1.2　内部有效性

在本文中，我们使用了两个流行的分析工具 sonarquobe 和 aDoctor 来分别提取 Java 静态代码指标和 Android 代码异味。这两个工具引入的一些不明显的错误可能在一定程度上威胁到我们的结果。此外，我们借助 Androrisk 工具为应用程序的风险划分等级。为了确保我们的标签是合理的，我们基于 Androrisk 的分析结果重新检查了相关文件，并对一些假阳性结果进行修复。然而，我们的修复方案也存在一些设计疏忽，我们不能说我们的标签百分之百正确。

6.1.3　外部有效性

在我们的研究中，使用的所有应用程序都来自 Github。我们不能保证我们的结论适用于其他开源软件平台或工业界。然而，考虑到我们的应用程序来自于不同的领域，具有不同的规模，我们相信我们的研究仍然给代码异味和应用程序的安全性研究提供了一些有价值的信息。

6.2　工作总结

当今互联网技术发展迅速，其中势头最猛的当属移动应用程序，它在人们的日常生活中发挥着重要的作用。特别明显的是，电子支付已经逐步大众化，人

们习惯于出门只带手机，而不带纸币或银行卡。消费者通常会依赖并信任应用程序，他们会为其提供一些隐私的、机密的数据。这样的敏感属性使得应用程序容易成为黑客的攻击目标，他们会利用应用中的漏洞窃取个人信息甚至是钱财，从而给用户带来难以预计的亏损。所以，针对移动应用的风险评估工作就显得越来越重要。开发者们在开发应用程序时，需要更好地评估应用程序质量，并针对不良的代码结构进行修改，从而降低应用程序的风险。在移动应用市场，安卓系统占领了 87% 的市场份额。特别是在中国的手机市场，安卓系统更是以绝对占比优势领先于其他应用系统。安卓系统最大的特点就是其系统开源，学习成本较低，系统迭代更新的速度快，依赖于很多第三方库。这些特点导致了安卓应用程序的开发者很多都是非专业型人才，他们往往缺少规范的代码编写能力，更容易引入不良的代码结构，从而造成应用程序的高风险。因此，如何使用自动化的方式对应用程序进行风险评估，是应用程序相关从业者比较关注的问题。

在此背景下，安卓应用程序的安全性和质量保障成为了我们最关注的问题。为了更好地评估安卓应用程序的风险程度，依托于 GitHub 上丰富多样的开源安卓应用程序，本文设计并实现了一套基于多维安卓特征的移动应用风险评估系统，以满足上述多种日常场景下对安卓应用程序的安全需求。本系统实现了高准确度的安卓代码异味检测工具，基于代码异味构建了应用程序的安全风险等级预测模型，同时构建了故障数模型，探索代码异味对安卓应用程序安全、性能、内存等各方面的影响。所以本系统可以满足现有业务需求和场景，帮助应用程序相关从业者及时发现代码中的不良结构，并评测应用程序的风险程度。

在本文中，我们首先介绍了与移动安全相关的研究背景及意义，分析了现有研究的一些局限性，并将本系统与现有系统作比较，阐述本系统的优势。其次，我们介绍了与本系统相关的代码指标，特别强调了安卓代码异味，分析了代码异味对软件质量的影响。其次，本文还介绍了系统所涉及的一些算法理论知识、开源检测工具，并解释了选择这些算法和工具的原因和理由。在第三章，本文特别增加了一个特征预处理章节，详细介绍了安卓特征的处理过程，描述了每一个处理步骤的细节和处理原因。在第四章，本文对移动应用风险评估技术展开了讨论，主要介绍了三个研究工作：（1）定义 15 种与安全相关的安卓代码异味，并实现了检测它们的自动化工具 DACS（2）构建安全风险等级预测模型以及（3）构建了故障数计数模型。对于工作一，本文将不良的代码结构和代码漏洞形式化，定义了 15 个全新的安卓代码异味，并对这些代码异味提出了修复意见。为了自动化检测这些异味，本文开发了工具 DACS。具体来说，DACS 将应用程序的源代码抽象成语法树，并基于代码异味的定义实现检测算法。通过不断搜索语法树，找出包含代码异味的代码片段。本文收集了 20 个开源的安

卓应用程序，邀请了志愿者来人工检测代码异味。同时，本文也利用 DACS 工具自动化检测这 20 个应用。通过比较两份结果的一致性，评估 DACS 工具的有效性。结果表明，DACS 工具可以有效代替人工完成代码异味检测任务。对于工作二，本文提出引入 Android 代码异味来帮助预测 Android 应用程序的安全风险等级。更具体地说，本文结合了 Android 代码异味和 Java 静态代码指标，并在这些混合指标上应用了多种机器学习方法来建立相关的预测模型。本文构建了一个来自 GitHub 上的 4575 个 Android 应用程序的数据集，以评估预测模型的有效性。结果发现随机森林学习算法优于其他学习方法，AUC 达到 0.97。此外，一些代码异味如非静态内部方法 (MIM) 和内部类泄露 (LIC) 在预测模型中的贡献度较大。对于工作三，本文挑选了五种代码异味（弱加密算法（WCA）异味，头文件（HA）异味，数据备份（DBA）异味，恶意压缩（MU）异味和 SD 卡访问（SDV）异味），探索了其对安卓应用程序安全、性能、内存等各方面的影响。具体来说，本文以代码异味和代码行数（NCLOC）指标为自变量，应用程序的故障数为因变量，并分别应用了负二项回归算法和零膨胀负二项回归算法，构建了故障数模型。结果表明，NCLOC，MU 和 WCA:NCLOC 会显著增加应用的故障倾向性，WCA，SDV 和 NCLOC:MU 会显著减少故障倾向性，HA 和 DBA 和故障倾向性无关。至此，本文完成了对安卓应用的详细风险评估工作。

6.3 工作展望

本系统为安卓应用程序的安全和规范保驾护航，并对不良的代码结构提供了有建设性的意见和修复方向。但是，如第6.1节所述，本文还存在着一些不足之处。今后，我们将从以下几个方面进行改进。

（1）Android 系统的更新迭代迅速，其代码的升级也会引入新的代码异味。此外，本系统所提出的代码异味有限，在未来，我们可以发掘更多类型的 Android 代码异味，进一步提高模型在预测应用程序风险上的准确性。

（2）本文实验的应用程序集是从 GitHub 平台上收集的，平台的局限性也限制了应用程序的多样性，我们计划扩展原先的应用程序数据集，从其他开放源码软件平台或工业界收集更多的安卓应用程序。

（3）本系统开发的 DACS 工具是根据规则匹配来检测代码异味的，规则库中收集了安卓系统中常用的第三方库和方法，但是在少量应用程序中，它们会引入不常见的方法，导致这些应用的代码异味会被遗检。未来，我们还要不断更新完善规则库，保证 DACS 工具不误报不漏报。

（4）目前系统针对不良的代码结构提供了相应的修复意见，但是在修复后是否能够真正提高应用程序的质量，这还需要后续实验的证明。

参考文献

[1] B. A. Kitchenham, Software quality assurance, Microprocessors and microsystems 13 (6) (1989) 373–381.

[2] R. Fulton, R. Vandermolen, Airborne Electronic Hardware Design Assurance: A Practitioner's Guide to RTCA/DO-254, CRC Press, 2014.

[3] S. L. Henry, S. Abou-Zahra, J. Brewer, The role of accessibility in a universal web, in: Proceedings of the 11th Web for all Conference, ACM, 2014, p. 17.

[4] A. Geraci, F. Katki, L. McMonegal, B. Meyer, J. Lane, P. Wilson, J. Radatz, M. Yee, H. Porteous, F. Springsteel, IEEE standard computer dictionary: Compilation of IEEE standard computer glossaries, IEEE Press, 1991.

[5] A. B. Bondi, Characteristics of scalability and their impact on performance, in: Proceedings of the 2nd international workshop on Software and performance, ACM, 2000, pp. 195–203.

[6] B. Buzan, O. Wæver, O. Wæver, J. De Wilde, Security: a new framework for analysis, Lynne Rienner Publishers, 1998.

[7] L. Cardelli, P. Wegner, On understanding types, data abstraction, and polymorphism, ACM Computing Surveys (CSUR) 17 (4) (1985) 471–523.

[8] R. Dawkins, Hierarchical organisation: A candidate principle for ethology, Growing points in ethology 7 (1976) 54.

[9] M. L. Scott, Programming language pragmatics, Morgan Kaufmann, 2000.

[10] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: improving the design of existing code, Addison-Wesley Professional, 1999.

[11] N. Tsantalis, T. Chaikalis, A. Chatzigeorgiou, Jdeodorant: Identification and removal of type-checking bad smells, in: Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on, IEEE, 2008, pp. 329–331.

[12] E. Murphy-Hill, A. P. Black, An interactive ambient visualization for code smells, in: Proceedings of the 5th international symposium on Software visualization, ACM, 2010, pp. 5–14.

[13] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, E. Figueiredo, A review-based comparative study of bad smell detection tools, in: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, ACM, 2016, p. 18.

[14] S. Olbrich, D. S. Cruzes, V. Basili, N. Zazworka, The evolution and impact of code smells: A case study of two open source systems, in: Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on, IEEE, 2009, pp. 390–400.

[15] S. M. Olbrich, D. S. Cruzes, D. I. Sjøberg, Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems, in: Software Maintenance (ICSM), 2010 IEEE International Conference on, IEEE, 2010, pp. 1–10.

[16] D. I. Sjoberg, A. Yamashita, B. C. Anda, A. Mockus, T. Dyba, Quantifying the effect of code smells on maintenance effort, IEEE Transactions on Software Engineering (8) (2013) 1144–1156.

[17] T. Hall, M. Zhang, D. Bowes, Y. Sun, Some code smells have a significant but small effect on faults, ACM Transactions on Software Engineering and Methodology (TOSEM) 23 (4) (2014) 33.

[18] M. Linares-Vásquez, S. Klock, C. McMillan, A. Sabané, D. Poshyvanyk, Y.-G. Guéhéneuc, Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps, in: Proceedings of the 22nd International Conference on Program Comprehension, ACM, 2014, pp. 232–243.

[19] D. Verloop, Code smells in the mobile applications domain, Ph.D. thesis, TU Delft, Delft University of Technology (2013).

[20] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, C. Jensen, Understanding code smells in android applications, in: Mobile Software Engineering and

Systems (MOBILESoft), 2016 IEEE/ACM International Conference on, IEEE, 2016, pp. 225–236.

[21] G. Hecht, R. Rouvoy, N. Moha, L. Duchien, Detecting antipatterns in android apps, in: Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems, IEEE Press, 2015, pp. 148–149.

[22] J. Reimann, M. Brylski, U. Aßmann, A tool-supported quality smell catalogue for android developers, in: Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung–MMSM, Vol. 2014, 2014.

[23] G. Hecht, N. Moha, R. Rouvoy, An empirical study of the performance impacts of android code smells, in: Proceedings of the International Conference on Mobile Software Engineering and Systems, ACM, 2016, pp. 59–69.

[24] M. Ghafari, P. Gadient, O. Nierstrasz, Security smells in android, in: 2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, 2017, pp. 121–130.

[25] M. Gottschalk, M. Josefiok, J. Jelschen, A. Winter, Removing energy code smells with reengineering services., GI-Jahrestagung 208 (2012) 441–455.

[26] G. Hecht, An approach to detect android antipatterns, in: Proceedings of the 37th International Conference on Software Engineering-Volume 2, IEEE Press, 2015, pp. 766–768.

[27] G. Hecht, L. Duchien, N. Moha, R. Rouvoy, Detection of anti-patterns in mobile applications, in: COMPARCH 2014, 2014.

[28] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, A. De Lucia, Lightweight detection of android-specific code smells: The adoctor project, in: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2017, pp. 487–491.

[29] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, L. Duchien, Tracking the software quality of android applications along their evolution (t), in: Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on, IEEE, 2015, pp. 236–247.

[30] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, A. De Lucia, On the impact of code smells on the energy consumption of mobile applications, Information and Software Technology 105 (2019) 43–55.

[31] A. Koenig, Patterns and antipatterns, The patterns handbook: techniques, strategies, and applications 13 (1998) 383.

[32] W. Enck, D. Octeau, P. D. McDaniel, S. Chaudhuri, A study of android application security., in: USENIX security symposium, Vol. 2, 2011.

[33] E. Chin, A. P. Felt, K. Greenwood, D. Wagner, Analyzing inter-application communication in android, in: Proceedings of the 9th international conference on Mobile systems, applications, and services, 2011, pp. 239–252.

[34] A. Rahman, P. Pradhan, A. Partho, L. Williams, Predicting android application security and privacy risk with static code metrics, in: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), IEEE, 2017, pp. 149–153.

[35] F. Qin, Z. Zheng, X. Li, Q. Yu, K. S. Trivedi, An empirical investigation of fault triggers in android operating system, in: 2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC), 2017.

[36] D. Cotroneo, A. K. Iannillo, R. Natella, S. Rosiello, Dependability assessment of the android os through fault injection, IEEE Transactions on Reliability PP (99) (2019) 1–16.

[37] H. Mirzaei, A. Heydarnoori, Exception fault localization in android applications, in: 2015 2nd ACM International Conference on Mobile Software Engineering and Systems, IEEE, 2015, pp. 156–157.

[38] Y. J. Ham, D. Moon, H.-W. Lee, J. D. Lim, J. N. Kim, Android mobile application system call event pattern analysis for determination of malicious attack, International Journal of Security and Its Applications 8 (1) (2014) 231–246.

[39] V. Svetnik, A. Liaw, C. Tong, J. C. Culberson, R. P. Sheridan, B. P. Feuston, Random forest: a classification and regression tool for compound classification and qsar modeling, Journal of chemical information and computer sciences 43 (6) (2003) 1947–1958.

[40] N. Nagappan, T. Ball, A. Zeller, Mining metrics to predict component failures, in: Proceedings of the 28th international conference on Software engineering, 2006, pp. 452–461.

[41] H. Alves, B. Fonseca, N. Antunes, Software metrics and security vulnerabilities: Dataset and exploratory study, in: Dependable Computing Conference (EDCC), 2016 12th European, IEEE, 2016, pp. 37–44.

[42] M. Alenezi, I. Almomani, Empirical analysis of static code metrics for predicting risk scores in android applications, in: 5th International Symposium on Data Mining Applications, Springer, 2018, pp. 84–94.

[43] T. Hall, S. Beecham, D. Bowes, D. Gray, S. Counsell, A systematic literature review on fault prediction performance in software engineering, IEEE Transactions on Software Engineering 38 (6) (2011) 1276–1304.

[44] J. Hryszko, L. Madeyski, M. Dąbrowska, P. Konopka, Defect prediction with bad smells in code, arXiv preprint arXiv:1703.06300.

[45] G. Succi, W. Pedrycz, M. Stefanovic, J. Miller, Practical assessment of the models for identification of defect-prone classes in object-oriented commercial systems using design metrics, Journal of systems and software 65 (1) (2003) 1–12.

[46] R. Moser, W. Pedrycz, G. 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, 2008, pp. 181–190.

[47] L. Madeyski, M. Jureczko, Which process metrics can significantly improve defect prediction models? an empirical study, Software Quality Journal 23 (3) (2015) 393–422.

[48] M. Jureczko, L. Madeyski, A review of process metrics in defect prediction studies, Metody Informatyki Stosowanej 5 (2011) 133–145.

[49] M. Zhang, T. Hall, N. Baddoo, Code bad smells: a review of current knowledge, Journal of Software Maintenance and Evolution: research and practice 23 (3) (2011) 179–202.

[50] M. Zhang, T. Hall, N. Baddoo, P. Wernick, Do bad smells indicate" trouble" in code?, in: Proceedings of the 2008 workshop on Defects in large software systems, 2008, pp. 43–44.

[51] T. Holschuh, M. Pauser, K. Herzig, T. Zimmermann, R. Premraj, A. Zeller, Predicting defects in sap java code: An experience report, in: 2009 31st International Conference on Software Engineering-Companion Volume, IEEE, 2009, pp. 172–181.

[52] W. H. Brown, R. C. Malveau, H. W. McCormick, T. J. Mowbray, AntiPatterns: refactoring software, architectures, and projects in crisis, John Wiley & Sons, Inc., 1998.

[53] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, D. Poshyvanyk, When and why your code starts to smell bad, in: Proceedings of the 37th International Conference on Software Engineering-Volume 1, IEEE Press, 2015, pp. 403–414.

[54] J. Oliveira, M. Viggiato, M. Santos, E. Figueiredo, H. Marques-Neto, An empirical study on the impact of android code smells on resource usage, in: International Conference on Software Engineering & Knowledge Engineering (SEKE), 2018.

[55] R. Minelli, M. Lanza, Software analytics for mobile applications–insights & lessons learned, in: Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on, IEEE, 2013, pp. 144–153.

[56] M. Brylski, Android smells catalogue, https://martinbrylski.github.io/android_smells (2013).

[57] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, R. Rouvoy, Investigating the energy impact of android smells, in: 24th International IEEE Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2017, p. 10.

[58] M. Kessentini, A. Ouni, Detecting android smells using multi-objective genetic programming, in: Proceedings of the 4th International Conference on Mobile Software Engineering and Systems, IEEE Press, 2017, pp. 122–132.

[59] N. Moha, Y.-G. Gueheneuc, A.-F. Duchien, et al., Decor: A method for the specification and detection of code and design smells, IEEE Transactions on Software Engineering (TSE) 36 (1) (2010) 20–36.

[60] 李炎武, 软件工程中代码异味检测方法的研究, 现代计算机 (05) (2017) 31–33.

[61] R. Marinescu, Detection strategies: Metrics-based rules for detecting design flaws, in: Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on, IEEE, 2004, pp. 350–359.

[62] 赵敏, 高建华, 结合遗传规划和遗传算法的代码异味检测方法, 小型微型计算机系统 v.41 (11) (2020) 196–203.

[63] 王继娜, 陈军华, 高建华, 基于排序损失的 ecc 多标签代码异味检测方法, 计算机研究与发展 58 (1) (2021) 178.

[64] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, D. Poshyvanyk, Detecting bad smells in source code using change history information, in: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, IEEE Press, 2013, pp. 268–278.

[65] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, M. S. Hamdi, Improving multi-objective code-smells correction using development history, Journal of Systems and Software 105 (2015) 18–39.

[66] N. Moha, Y.-G. Guéhéneuc, A.-F. Le Meur, L. Duchien, A. Tiberghien, From a domain analysis to the specification and detection of code and design smells, Formal Aspects of Computing 22 (3-4) (2010) 345–361.

[67] N. Tsantalis, A. Chatzigeorgiou, Identification of extract method refactoring opportunities for the decomposition of methods, Journal of Systems and Software 84 (10) (2011) 1757–1782.

[68] F. A. Fontana, E. Mariani, A. Mornioli, R. Sormani, A. Tonello, An experience report on using code smells detection tools, in: Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on, IEEE, 2011, pp. 450–457.

[69] M. Delchev, M. F. Harun, Investigation of code smells in different software domains, Full-scale Software Engineering 31.

[70] Ġ. A. SAĞLAM, Measuring and assesment of well known bad practices in android application developments, Ph.D. thesis, MIDDLE EAST TECHNICAL UNIVERSITY (2014).

[71] Y. Zhou, H. Leung, B. Xu, Examining the potentially confounding effect of class size on the associations between object-oriented metrics and change-proneness, IEEE Transactions on Software Engineering 35 (5) (2009) 607–623.

[72] C. Gibler, J. Crussell, J. Erickson, H. Chen, Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale, in: International Conference on Trust and Trustworthy Computing, Springer, 2012, pp. 291–307.

[73] B. V. Chess, Improving computer security using extended static checking, in: Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on, IEEE, 2002, pp. 160–173.

[74] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, K.-P. Wu, Droidmat: Android malware detection through manifest and api calls tracing, in: Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on, IEEE, 2012, pp. 62–69.

[75] A. Kumar, K. P. Sagar, K. Kuppusamy, G. Aghila, Machine learning based malware classification for android applications using multimodal image representations, in: Intelligent Systems and Control (ISCO), 2016 10th International Conference on, IEEE, 2016, pp. 1–6.

[76] A. Bose, X. Hu, K. G. Shin, T. Park, Behavioral detection of malware on mobile handsets, in: Proceedings of the 6th international conference on Mobile systems, applications, and services, ACM, 2008, pp. 225–238.

[77] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, Y. Weiss, "andromaly": a behavioral malware detection framework for android devices, Journal of Intelligent Information Systems 38 (1) (2012) 161–190.

[78] K. Dunham, S. Hartman, M. Quintans, J. A. Morales, T. Strazzere, Android Malware and Analysis, Auerbach Publications, 2014.

[79] 张艳, 任子晖, 一类基于支持向量机的软件故障预测方法, 小型微型计算机系统 31 (7) (2010) 1380–1384.

[80] O. Mizuno, S. Ikami, S. Nakaichi, T. Kikuno, Spam filter based approach for finding fault-prone software modules, in: International Workshop on Mining Software Repositories, 2007.

[81] O. Mizuno, T. Kikuno, Training on errors experiment to detect fault-prone software modules by spam filter, in: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, 2007, pp. 405–414.

[82] Y. Zhou, B. Xu, H. Leung, On the ability of complexity metrics to predict fault-prone classes in object-oriented systems, Journal of Systems and Software 83 (4) (2010) 660–674.

[83] H. Zhang, An investigation of the relationships between lines of code and defects, in: 2009 IEEE International Conference on Software Maintenance, IEEE, 2009, pp. 274–283.

[84] T. J. Ostrand, E. J. Weyuker, R. M. Bell, Where the bugs are, ACM SIGSOFT Software Engineering Notes 29 (4) (2004) 86–96.

[85] R. M. Bell, T. J. Ostrand, E. J. Weyuker, Looking for bugs in all the right places, in: Proceedings of the 2006 international symposium on Software testing and analysis, 2006, pp. 61–72.

[86] Y. Shin, R. Bell, T. Ostrand, E. Weyuker, Does calling structure information improve the accuracy of fault prediction?, in: 2009 6th IEEE International Working Conference on Mining Software Repositories, IEEE, 2009, pp. 61–70.

[87] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, B. Murphy, Change bursts as defect predictors, in: 2010 IEEE 21st International Symposium on Software Reliability Engineering, IEEE, 2010, pp. 309–318.

[88] M. D'Ambros, M. Lanza, R. Robbes, On the relationship between change coupling and software defects, in: 2009 16th Working Conference on Reverse Engineering, IEEE, 2009, pp. 135–144.

[89] T. J. Ostrand, E. J. Weyuker, R. M. Bell, Programmer-based fault prediction, in: Proceedings of the 6th International Conference on Predictive Models in Software Engineering, 2010, pp. 1–10.

[90] C. Bird, N. Nagappan, H. Gall, B. Murphy, P. Devanbu, Putting it all together: Using socio-technical networks to predict failures, in: 2009 20th International Symposium on Software Reliability Engineering, IEEE, 2009, pp. 109–119.

[91] S. Shivaji, E. J. Whitehead Jr, R. Akella, S. Kim, Reducing features to improve bug prediction, in: 2009 IEEE/ACM International Conference on Automated Software Engineering, IEEE, 2009, pp. 600–604.

[92] P. L. Gupta, R. C. Gupta, R. C. Tripathi, Analysis of zero-adjusted count data, Computational Statistics & Data Analysis 23 (2) (1996) 207–218.

[93] M. Greenwood, G. U. Yule, An inquiry into the nature of frequency distributions representative of multiple happenings with particular reference to the occurrence of multiple attacks of disease or of repeated accidents, Journal of the Royal statistical society 83 (2) (1920) 255–279.

[94] C. D. Kemp, N. L. Johnson, S. Kotz, Distribution in statistics: Discrete distributions, Revue de lÏnstitut International de Statistique / Review of the International Statistical Institute 39 (2) (1971) 243.

[95] D. Lambert, Zero-inflated poisson regression, with an application to defects in manufacturing, Technometrics 34 (1) (1992) 1–14.

[96] W. Greene, Accounting for excess zeros and sample selection in poisson and negative binomial regression models (1994) EC–94–10.

[97] A. S. Andreou, S. P. Chatzis, Software defect prediction using doubly stochastic poisson processes driven by stochastic belief networks, Journal of Systems and Software 122 (2016) 72–82.

[98] A. C. Cameron, P. K. Trivedi, Regression analysis of count data, Vol. 53, Cambridge university press, 2013.

[99] S. S. Rathore, S. Kumar, Towards an ensemble based system for predicting the number of software faults, Expert Systems with Applications 82 (2017) 357–382.

[100] T. Sethi, et al., Improved approach for software defect prediction using artificial neural networks, in: Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO), 2016 5th International Conference on, IEEE, 2016, pp. 480–485.

[101] K. Gao, T. M. Khoshgoftaar, A comprehensive empirical study of count models for software fault prediction, IEEE Transactions on Reliability 56 (2) (2007) 223–236.

[102] T. M. Khoshgoftaar, K. Gao, N. Seliya, Attribute selection and imbalanced data: Problems in software defect prediction, in: 2010 22nd IEEE International conference on tools with artificial intelligence, Vol. 1, IEEE, 2010, pp. 137–144.

[103] L. Al Shalabi, Z. Shaaban, B. Kasasbeh, Data mining: A preprocessing engine, Journal of Computer Science 2 (9) (2006) 735–739.

[104] V. R. Patel, R. G. Mehta, Impact of outlier removal and normalization approach in modified k-means clustering algorithm, International Journal of Computer Science Issues (IJCSI) 8 (5) (2011) 331.

[105] N. K. Visalakshi, K. Thangavel, Impact of normalization in distributed k-means clustering, international Journal of Soft computing 4 (4) (2009) 168–172.

[106] G. W. Milligan, M. C. Cooper, A study of standardization of variables in cluster analysis, Journal of classification 5 (2) (1988) 181–204.

[107] N. E. Fenton, M. Neil, A critique of software defect prediction models, IEEE Transactions on software engineering 25 (5) (1999) 675–689.

[108] N. V. Chawla, K. W. Bowyer, L. O. Hall, W. P. Kegelmeyer, Smote: synthetic minority over-sampling technique, Journal of artificial intelligence research 16 (2002) 321–357.

[109] J. King, Android application security with owasp mobile top 10 2014, https://owasp.org/www-project-mobile-top-10 (2014).

[110] M. Syakur, B. Khotimah, E. Rochman, B. Satoto, Integration k-means clustering method and elbow method for identification of the best customer profile cluster, in: IOP Conference Series: Materials Science and Engineering, Vol. 336, IOP Publishing, 2018, p. 012017.

[111] G. E. A. P. A. Batista, R. C. Prati, M. C. Monard, A study of the behavior of several methods for balancing machine learning training data, Acm Sigkdd Explorations Newsletter 6 (1) (2004) 20–29.

[112] S. S. Rathore, S. Kumar, A study on software fault prediction techniques, Artificial Intelligence Review (2017) 1–73.

[113] T. Hall, D. Bowes, G. Liebchen, P. Wernick, Evaluating three approaches to extracting fault data from software change repositories, in: International Conference on Product Focused Software Process Improvement, Springer, 2010, pp. 107–115.

[114] S. Kim, T. Zimmermann, E. J. Whitehead Jr, A. Zeller, Predicting faults from cached history, in: Proceedings of the 29th international conference on Software Engineering, IEEE Computer Society, 2007, pp. 489–498.

[115] I. Lawrence, K. Lin, A concordance correlation coefficient to evaluate reproducibility, Biometrics (1989) 255–268.

[116] D. G. Bonett, T. A. Wright, Sample size requirements for estimating pearson, kendall and spearman correlations, Psychometrika 65 (1) (2000) 23–28.

[117] G. Czibula, Z. Marian, I. G. Czibula, Software defect prediction using relational association rule mining, Information Sciences 264 (2014) 260–278.

[118] C. Leistner, A. Saffari, J. Santner, H. Bischof, Semi-supervised random forests, in: 2009 IEEE 12th International Conference on Computer Vision, IEEE, 2009, pp. 506–513.

[119] R. Liu, J. Cheng, H. Lu, A robust boosting tracker with minimum error bound in a co-training framework, in: 2009 IEEE 12th International Conference on Computer Vision, IEEE, 2009, pp. 1459–1466.

[120] B. Zhang, G. Ye, Y. Wang, J. Xu, G. Herman, Finding shareable informative patterns and optimal coding matrix for multiclass boosting, in: 2009 IEEE 12th International Conference on Computer Vision, IEEE, 2009, pp. 56–63.

简历与科研成果

基本情况 龚爱，女，汉族，1996 年 4 月出生，江苏省扬州市人。

教育背景

2018.9 ~ 2021.6 南京大学软件学院 硕士

2014.9 ~ 2018.7 河海大学物联网工程学院 本科

读研期间的成果

1. **Ai Gong**, Yi Zhong, Weiqin Zou, Yangyang Shi, Chunrong Fang，“Incorporating Android Code Smells into Java Static Code Metrics for Security Risk Prediction of Android Applications”，The 20th International Conference on Software Quality, Reliability and Security. QRS, 2020.
2. 房春荣, **龚爱**, 王加琪, 陈振宇, “一种融合时间序列算法和排序算法的民航机票代理市场的预测方法”, 专利申请号:202011028653.2, 2020.
3. 房春荣, **龚爱**, 王栋, 陈振宇, 李玉莹, “一种基于噪音数据的机器学习模型鲁棒性评估方法”, 专利申请号:2019109520574, 2019.
4. 房春荣, **龚爱**, 徐文远, 蒋燕, 李玉莹, 陈振宇, “一种基于 Android 源代码的风险等级预测方法”, 专利申请号:201810092876.1, 2018.

致　　谢

在毕业设计完成之际，我想要对研究生三年以来帮助我、陪伴我的老师同学们表达衷心的感谢！

首先要感谢的是我的研究生导师陈振宇老师，是陈老师对我的认可和支持才让我有机会进入 ISE 实验室继续学习深造。研究生期间，陈老师在科研上给了我很多宝贵的建议和具体的指导。陈老师知识渊博，治学严谨，思路开阔，在他的引导下，我接触了一些机器学习理论知识，开阔了我的视野，为本文的撰写打下了基础。除了学习上的指导外，陈老师还在生活上和思想上关心我，教会我做人做事的方法和原则。陈老师幽默风趣的教学风格，认真勤奋的工作作风，以及积极向上的生活态度将会激励我在以后的工作生活中奋发图强，永不言弃。

其次要感谢的是我的实验室导师以及毕业设计指导老师房春荣老师。在我初次发表论文时，总是因为各种原因屡次被拒，信心受挫。房老师却一直鼓励我，陪我讨论研究方案，指导我修改论文，最终才能将论文顺利发出。此外，在毕业论文撰写阶段，因为疫情的原因我们只能在家完成论文的写作工作。虽然条件苛刻，但是房老师还是通过各种途径，关心我的毕设进展，定期检查我的进度，以他丰富的专业知识和科研经验，解决了我在毕业设计中遇到的各种问题，至此，我想对房老师表达我最真挚的感谢！

同时，我还要感谢我的三位研究生舍友，沈思媛，曹楠和蔡莹月。每当我结束一天的科研工作回到宿舍时，迎接我的总是欢声和笑语。大家在一起总是悲伤减半，快乐加倍。很开心能和他们度过这美好的时光，这将成为我人生中最宝贵的回忆。我还要感谢 ISE 实验室 2018 级的朋友们。他们大部分都已经在去年毕业，但是他们还是给予了我很多的帮助，特别是黎宇和李林昱，他们总是会以“过来人”的身份给我很多的建议和帮助，让我能少走很多弯路，更顺利的毕业。

最后要感谢的是我的父母和伴侣，感谢父母这二十多年来陪伴我、帮助我、指引我、教育我，让我能快乐健康的长大。也要感谢我的伴侣，谢谢他一直以来给予我的理解、支持和鼓励。家人们的的关怀和爱让我体会到了生活的美好，他们就是我不断努力进步的永恒动力。