# Kotlin语言编写的Android应用程序质量的实证研究

**摘要**

**背景：**在过去的几年中，移动应用程序的开发者有可能使用新的范例和工具来开发移动应用程序。例如，自2017年以来，Android开发者获得了使用Kotlin语言编写Android应用程序的官方支持。Kotlin是一种与Java完全互操作的编程语言，它结合了面向对象和功能特性。

**目的**：本文的目标是双重的。首先，它旨在研究Kotlin语言在开源Android应用程序开发中的采用程度，并测量这些应用程序中Kotlin代码的数量。其次，它旨在衡量使用Kotlin编写的Android应用程序的质量，并将其与使用Java编写的Android应用程序的质量进行比较。

**方法：**我们首先定义了一种从开源Android应用程序的数据集中检测Kotlin应用程序的方法。然后，我们分析这些应用程序来检测代码异味的实例，并计算应用程序的质量评估。最后，我们研究了Kotlin代码的引入如何影响Android应用程序的质量。

**结果：**我们的实验发现，在包含2167个开源应用的数据集中，有11.26%的应用是使用Kotlin语言编写的(部分或全部)。我们发现，以所研究的10种不同代码异味（其中4种是面向对象的，6种是Android的）为考核标准，对于大多数最初用Java编写的Android应用程序而言， Kotlin代码的引入提高了应用程序的质量。

关键词：Android；移动开发；代码质量；Kotlin；Java；代码异味；开源应用

## 导言

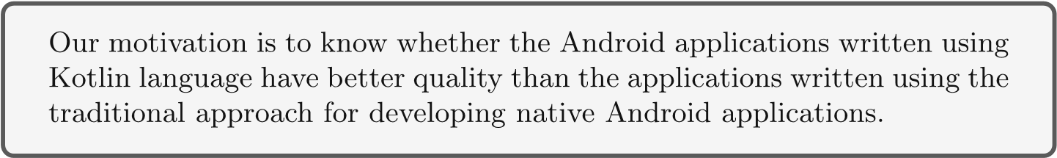
2017年，智能手机公司共出货14.6亿部设备。他们中的绝大多数(85%)运行一个平台，即谷歌的Android平台，该平台比第二大使用平台苹果的iOS平台更受欢迎，占全球智能手机量的14.7%。

为了开发能够在装有Android的设备上运行的移动应用程序，Google提供了一个官方IDE(集成开发环境)，名为Android Studio，和SDK(软件开发工具包)，Android允许运行最初用Java编写的原生应用程序。SDK将Java代码编译成Dalvik字节码，然后打包成apk。然后，开发者将这些apk提交给应用商店，比如官方的谷歌Play商店。Android用户可以通过直接从应用商店下载这些apk来将应用安装到他们的Android设备中。这些Android设备包括一个能够运行apk的虚拟机。

在过去几年中，出现了不同的开发方法和框架来简化移动应用程序的开发。例如，来自Adobe的PhoneGap/Cordova等方法，来自Microsoft的Xamarin，来自FaceBook的React-Native。他们都旨在通过允许开发人员使用非本地编程语言编写应用程序，然后获得每个平台(Android和iOS)的版本，来促进多平台移动应用程序的开发。

与此同时，谷歌和苹果继续发展他们的开发工具包来构建本地应用程序，目的是避免移动开发者迁移到第三方开发框架，如来自他们的竞争对手Microsoft的Xamarin Visual Studio。2014年6月，苹果发布了Swift，这是一种现代的多范式语言，结合了命令式、面向对象和函数式编程，用于开发iOS应用程序。

2017年，谷歌宣布Kotlin编程语言作为Android开发的官方支持语言。Kotlin是一种实用的编程语言，运行在Java虚拟机和Android上。它结合了面向对象和函数式编程，并且完全可以与Java互通。因此，可以在同一个应用程序中混合Kotlin和Java代码，从Java代码中调用Koltin代码，反之亦然。安卓官方博客指出，截至2017年底，Kotlin已在Android Studio 3.0中超过17%的项目中使用。

本文的目标是双重的。首先，研究Kotlin语言在Android应用程序上的采用情况以及在这些应用程序上编写的Kotlin代码量。其次，衡量使用Kotlin编写的Android应用程序的质量。

先前的几项研究调查了移动应用程序的质量，重点关注代码异味的存在，也称为反模式、能源消耗、性能，以及混合应用。然而，据我们所知，既没有研究采用Kotlin语言开发Android应用程序，也没有研究使用Kotlin语言编写的Android应用程序的质量。

为了进行我们的实验，我们首先建立一个开源Android应用的数据集。然后，我们分析这些应用程序的代码，计算每个应用程序中包含的Kotlin代码量。我们还研究了Kotlin和Java代码的数量是如何随着Android应用程序的历史演变的。之后，我们关注Android应用程序的质量。正如Hecht等人(2015a)所提议的，我们根据代码异味的存在来衡量Android应用程序的质量。我们首先进行了一个实验，用于检测文献(Hecht等人，2015a，2016a)提出的十种代码异味的实例。我们比较了两组Android应用程序的代码异味:一组包含使用Koltin编写的应用程序，另一组包含使用Java编写的应用程序。最后，我们研究Kotlin代码的引入如何影响最初用Java编写的Android应用程序的质量。

本文的研究以以下研究问题为指导。

* **RQ 1:***在我们的开源应用程序数据集中，Kotlin在移动开发中的采用程度如何？*

我们考察了三个不同的Android应用数据集(F-Droid,AndroidTimeMachine以及Androzoo)，筛选了至少部分使用Kotlin构建的应用程序。

***发现1:***来自我们数据集的11.26%的开源Android应用程序包含Kotlin代码(即2167个中的244个)；

* **RQ 2:***Kotlin代码在移动应用中的比例是多少？*

我们分析了每一个被过滤的应用程序，并计算了用Kotlin语言编写的代码的比例。

***发现2:*** 33.61%的包含Kotlin的应用程序(即244个中的82个)完全是用Kotlin编写的，也就是说，那些不包含Java代码。

* **RQ 3:***引入Kotlin代码后，代码如何沿着一个Android应用程序的历史演变？*

我们计算了Android应用程序每次提交的Kotlin代码量，以分析Kotlin代码在历史上的演变趋势。

***发现3:***对于63.9%用Kotlin编写的应用程序(全部或部分)，Kotlin代码的数量随着发展而增加，而Java代码的数量却减少了。

* **RQ 4:***关于研究的代码异味，Kotlin和Java Android应用程序在代码异味存在方面有区别吗？*

我们复制了Habchi等人(2017年)所做的实验，该实验比较了iOS和Android应用程序，以比较在两个Android应用程序数据集中发现的代码异味:一个有Kotlin代码，另一个没有任何Kotlin代码。

***发现4:***对于4个面向对象代码中的3个，Kotlin比Java应用程序受影响更大，但是对于所有面向对象的代码，Java应用程序受统计相关性影响的实体平均更多。

* **RQ 5:***Kotlin的引入对Android应用程序版本质量的积极影响有多频繁？*

我们首先根据Hecht等人(2015a)的质量模型计算了Kotlin应用程序每个版本的质量。然后，我们测量了在最初用Java编写的应用程序中引入Kotlin代码对质量的影响。

***发现5:***在最初用Java编写的Android应用程序中引入Kotlin代码，至少提高了50%应用程序的质量。

本文的贡献在于:

* 检测用Kotlin编写的应用程序的方法；
* 244个开源Android应用的数据集，部分或全部使用Kotlin；
* 关于Java和Koltin代码数量随Android应用程序发展的研究；
* 一项比较Kotlin和Java应用程序中代码异味的研究；
* Android应用程序中引入Kotlin代码对软件质量影响的测量研究。

本文继续如下。第2节介绍了相关的工作。第3节描述了Kotlin和使用Kotlin语言开发Android应用程序。第4节介绍了用于回答研究问题的方法。第5节介绍了研究结果和对研究问题的回答。第6节概述了有效性的威胁。第7节对我们的结果进行了讨论。第八部分是论文的结论。

本文中提供的所有数据都在我们的附录中公开提供：https://github.com/UPHF/[kotlinandroid](https://github.com/UPHF/kotlinandroid) [.](https://github.com/UPHF/kotlinandroid)

## 相关工作

Kent Beck在识别代码中的质量问题时创造了术语“代码异味”,这些问题可以通过重构来提高软件的可维护性(Fowler et al. 1999)。从那时起，软件工程社区已经探索了各种相关的维度，包括提出一个异味目录，使用各种技术检测异味，探索异味之间的关系，以及识别异味的原因和影响(Sharma和Spinellis 2018)。然而，根据Aniche等人(2017)的说法，传统的代码异味捕捉到了良好设计的非常普遍的原则。此外，他们认为需要特定类型的代码异味来捕捉基于特定平台、架构或技术的软件系统上的“不良实践”。在这种背景下，有可能找到专注于对象关系映射框架(Chen et al. 2014)、Android应用程序(Verloop 2013赖曼等人，2014年b；Hecht等人2015b)、级联样式表(CSS)(mazi nian等人2014)和模型视图控制器(MVC)架构(Aniche等人2017)。

在这一节中，我们将讨论移动应用程序中代码异味检测和修复的相关文献，以及软件发展的相关工作。

### 检测移动应用程序上的反模式(代码异味)

Mannan等人(2016)比较了500个Android应用程序和750个Java桌面应用程序中众所周知的面向对象代码异味的存在。他们的结论是，这两种类型的应用程序在代码异味的密度方面没有太大的区别。然而，他们观察到，Android应用程序上代码异味的分布比桌面应用程序更加多样化。(Khalid等人，2016年)进行了一项关于代码异味的存在与应用程序评级之间关系的研究。他们在谷歌Play商店分析了10，000个Android应用程序及其评论，发现三类警告与最终用户评论中的投诉之间存在对应关系。

Reimann等人(2014年b)提出了一个30种优质异味的目录，专门用于Android。这些代码异味主要来源于Android官方文档中在线记录的好的和坏的实践，或者由开发人员在博客上报告他们的经验。Hecht等人(2015a，b)提出了一种名为Paprika的工具化方法，用于从移动应用程序的二进制文件(apk)中识别面向对象和Android特定的代码异味。Palomba等人(2017年)提出了一种检测工具，称为aDoctor，使用静态分析代码技术检测15种Android代码异味。他们在18个Android应用程序的测试床上测试了aDoctor，并获得了接近100%的检测精度。

Mannan等人(2016年)和Khalid等人(2016年)专注于检测移动应用中面向对象的异味，而Reimann等人(2014年b)提出了一个专用于Android的异味目录。此外，Hecht等人(2015a，b)和Palomba等人(2017)提出了识别Android异味的工具。

据我们所知，没有人研究过用Kotlin编写的Android应用程序的代码异味。

### 代码进化

自从Chidamber和Kemerer (1994)定义了面向对象的度量标准之后，面向对象的度量标准在评估软件质量方面越来越受欢迎。

李等(2017a)对8个典型开源移动应用的生命周期进行了实证研究，以更好地理解移动应用的发展。他们的结果表明，雷曼定律的子集(雷曼1980年)仍然适用于开源移动应用程序。此外，他们发现移动应用程序的增长并不平稳，随着第三方库的加入，软件的不稳定性也在增加。

据我们所知，还没有人研究过Kotlin应用程序的发展。

### 贯穿软件发展的代码异味(反模式)的存在

评估软件质量的其他工作集中在代码异味的存在上。

Palomba等人(2015)提出了HIST(异味检测的历史信息)，这是一种旨在通过利用从版本控制系统中提取的信息来检测五种不同代码异味的方法。他们基于对单个项目快照的分析，将他们的方法与手动构建的嗅觉神谕进行了比较，该嗅觉神谕是在20个Java开源项目中与传统方法进行识别的。他们发现，HIST可以识别代码异味，而竞争方法仅基于对单个系统快照的代码分析无法识别这些异味。此外，他们还证实了将历史和结构信息结合起来实现更好的异味检测的潜力。

Tufano等人(2015年)对来自不同软件生态系统(包括Android应用程序)的200个开源项目的变化历史进行了一项规模可观的实证研究。他们调查了开发商何时引入了难闻的异味，以及引入异味背后的环境和原因。他们发现，大多数时候，代码工件在创建后会受到代码异味的影响。此外，他们观察到，开发者倾向于引入异味的主要活动是实现新功能和增强现有功能。

Hecht等人(2015a，b)提出了一种工具化的方法，称为Paprika，用于从移动应用程序的二进制文件中识别面向对象和Android特定的反模式，并分析应用程序的质量及其演变。他们考虑了106个内部属性不同的Android应用程序，例如从最终用户的角度来看它们的大小和外部属性。他们收集了每个应用程序的几个版本(apk ),形成总共3568个版本，以评估软件在其发展过程中的质量。他们工作的一个主要发现是，移动应用程序开发人员需要分配更多的质量保证工作。Palomba等人(2018年)提出了一项关于代码异味的扩散及其对代码变更和错误倾向的影响的大规模实证研究。他们分析了30个开源项目的总共395个版本，考虑了13种不同的代码异味。他们发现，以长的和/或复杂的代码(例如，复杂的类)为特征的异味是高度扩散的，并且有异味的类比没有异味的类具有更高的变化和错误倾向。

因此，Tufano等人(2015)和Palomba等人(2015)专注于代码异味，但他们都没有调查软件进化过程中Android特有异味的存在。另一方面，Hecht等人(2015a，b)提出了一种能够识别异味类型的工具，面向对象和Android，他们评估了软件质量的演变。

然而，这些工作都没有研究用Kotlin编写的移动应用程序中异味的存在。

### 移动应用程序中代码异味和编程语言之间的关系

Habchi等人(2017)研究了iOS生态系统中的代码异味，考虑了Swift和Objective-C语言，以及它与Android异味的比较。他们提出了一个目录，列出了6种iOS特有的代码异味，这些异味是他们从开发者的反馈和平台官方那里确定的。为了识别这些代码异味，他们扩展了辣椒粉Hecht等人。然后，他们分析了GitHub上托管的103个Objective-C应用程序和176个Swift应用程序，发现在Objective-C和Swift中，代码异味往往以相同的比例出现，或者只有轻微的差异，并且用Objective-C和Swift编写的应用程序在面向对象的指标方面非常不同。此外，他们分析了F-Droid的1551个Android开源应用程序，发现在两种语言中，Android应用程序往往比iOS应用程序包含更多的代码异味，但SAK代码异味除外，它在所有语言中的出现比例相同。

虽然Habchi等人(2017)关注的是移动应用环境中代码异味和编程语言之间的关系，但他们没有考虑用Kotlin编写的应用。

### 测量代码异味对运行时的影响的研究

除了一些致力于识别代码异味的工作，其他作者也致力于更好地理解代码异味对运行时的影响。

Hecht等人(2016b)对不同版本的开源Android应用进行了一项实证研究，以确定修复Android反模式是否会对用户界面(延迟帧数)和内存使用产生重大影响。他们报告说，纠正这些Android代码异味有效地改善了用户界面，并以一种显著的方式提高了内存使用率。

Cruz和Abreu (2017)研究了八种基于性能的最佳实践对Android应用程序能耗的影响。他们对F-droid的6个应用程序进行了实证研究，并对这些应用程序中检测到的每个模式进行了手动重构。他们发现，固定ViewHolder、DrawAllocation、WakeLock、ObsoleteLayout-Param和Recycle可以提高能效。然而，固定未使用的资源和使用不透明并没有提供任何显著的能源消耗变化，而固定过度消耗增加了2.2%的能源消耗。

Saborido等人(2018)对Java SDK提供的实现HashMap和Android API提供的地图数据结构的不同实现进行了比较。他们分析了GitHub上可用的和Google Play上发布的5713个Android应用程序。他们发现，当键是基本类型时，应该使用SparseArray变体而不是HashMap和ArrayMap，因为它在CPU时间、内存和能耗方面更有效，这与Android文档部分不同。

这些研究集中在面向对象和Android的异味，以及它们如何影响不同的运行时方面。然而，这些实验没有考虑科特林的应用。

### 自动纠正代码异味的方法

之前的工作来自Mannan等人(2016)、Khalid等人(2016)、Li等人(2017b)、Palomba等人(2017)、Hecht等人(2015a，b，2016 b；Habchi等人2017)；克鲁兹和阿布雷乌(2017)；Saborido等人(2018)专注于研究和检测代码异味。其他的工作集中在应用“纠正”动作来克服代码异味。

Carette等人(2017年)提出了一种工具化和可重复的方法，称为Hot-Pepper，用于自动纠正代码异味并评估对能耗的影响。他们对五个Android应用程序进行了实证研究，以评估对能耗的影响。他们的结果证实了Android反模式对应用程序的能耗有影响。

Morales等人(2016，2017)介绍了一种在控制能耗的同时重构移动应用的新方法，名为EARMO，它使用了进化多目标技术。他们在20个免费和开源Android应用程序的基准上对其进行了评估。结果表明，EARMO可以提出平均84%的反模式的解决方案，平均执行时间不到一分钟。关于重构后的能耗差异，他们观察到三个应用程序改善了能耗，取得了具有统计意义的结果。

Cruz和Abreu (2018)提出了一种工具，能够识别和应用五种能量代码异味的自动重构:视图保持器、绘制分配、唤醒锁、回收和ObsoleteLayoutParam (Cruz和Abreu 2017)。他们分析了从F-droid收集的140个免费和开源的Android应用程序。他们的实验在45个应用程序中总共产生了222个重构，这些重构作为PRs提交给了最初的存储库。18个应用程序成功合并了它们的PR。这些研究(Morales等人2016，2017；克鲁兹和阿布雷乌2017，2018；Carette et al. 2017)在Java代码级别应用了重构来去除代码异味。

据我们所知，没有研究关注在Kotlin应用程序上应用重构。

## 使用Kotlin语言进行移动开发的简介

在本节中，我们将简要介绍使用 Kotlin 进行 Kotlin 和 Android 应用程序开发。

### Kotlin编程语言

Kotlin是一种静态类型的编程语言，相对于Java，它减少了语言的冗长性(例如，分号作为语句结束符“；”是可选的)).此外，Kotlin应用程序运行在Java虚拟机(JVM)之上。Kotlin编译器kotlinc将Kotlin代码编译成Java字节码，可以由JVM执行。出于这个原因，Kotlin也可以与Java和其他JVM语言(例如Scala或Groovy)互操作。这种互操作性可以从两个方面来看。首先，Kotlin开发人员可以使用用另一种语言(如Java)编写的库(如jar ),其次，开发人员可以使用Kotlin和其他JVM语言创建应用程序。

### Kotlin的新功能

Kotlin和Java是运行在JVM之上的现代编程语言。然而，与Java相比，Kotlin有一些不同之处。在我们看来，最相关的是，除了面向对象的范例之外，Kotlin还引入了函数式编程，例如，它允许使用高阶函数。

根据官方文件，下面列出了Kotlin的一些Java中没有的特性: 1）Lambda表达式和内联函数的组合，2)扩展函数，3)空安全，4)智能强制转换，5)属性，6)主构造函数，7)一级委托，8)变量和属性类型的类型推断，9)单例，10)声明站点差异和类型预测，11)范围表达式，12)运算符重载，13)伴随对象，14)数据类，15)只读和可变集合的独立接口，以及16) 协程

此外，根据提到的文档，Kotlin修复了Java遭受的一系列问题:1)空引用由类型系统控制，2)没有原始类型，3)Kotlin中的数组是不变的，4) Kotlin具有适当的函数类型，与Java的SAM-conversions相反，5)没有通配符的Use-site variance，以及6) Kotlin没有检查异常。

### 用于移动开发的Kotlin

要使用Kotlin开发移动应用程序，开发人员可以使用Google为使用Java开发Android应用程序提供的相同工具。这些工具是:a)软件开发工具包(SDK)，以及b)官方的Android集成开发环境(IDE)。例如，Android Studio 3.0+完全支持使用Kotlin代码开发Android应用程序，并提供自动完成、调试、重构和lint检查等功能。使用Android Studio，移动开发人员可以:1)启动一个新的Android项目，从头开始使用Kotlin开发应用程序，2)向已用Java编写的现有项目添加新的Kotlin文件，或3)将现有Java代码转换为Kotlin。

Android应用程序的源代码(用Java、Kotlin或两种语言编写)被编译成Dalvik字节码，它运行在为Android设备改编的JVM之上，名为Dalvik或Art(根据Android版本)。Dalvik字节码存储在。dex文件(类似于。JVM的类文件)。此外，Dalvik字节码被打包在一个名为Application Package Kit (APK)的文件中，该文件将移动应用程序的字节码分组，类似于Java中类文件的jar文件。Android SDK提供了使用名为apkanalyzer的工具检查这些文件的功能。

要生成apk，每个应用程序项目都必须在项目源集的根目录下有一个AndroidManifest.xml文件。清单文件描述了应用程序的基本信息，由Android构建工具、Android操作系统和Google Play使用。清单必须包括:应用程序的包名、应用程序的组件(活动、服务、广播接收器和内容提供者)、应用程序访问系统或其他应用程序的受保护部分所需的权限等。

### Kotlin的优势

据我们所知，没有人研究过使用Kotlin的优势。然而，Kotlin开发者社区列举了一些有利于Kotlin的特性，建议移动开发者应该从Java迁移到Kotlin。例如，Vinther (2017)提出了17个原因1) Java互操作性，2)熟悉的语法，3)字符串插值，4)类型推断，5)智能转换，6)直观的equals，7)默认参数，8)命名参数，when表达式，10)属性，11)数据类，12)运算符重载，13)析构声明，14)范围，15)扩展函数，16)空安全，以及17)更好的lambdas。据我们所知，还没有工作在移动开发的背景下从经验上验证这些原因。

## 方法论

在这一节中，我们将介绍本文中用于研究用Kotlin编写的Android应用程序的方法。第4.1节描述了从不同来源收集Android应用程序的方法，并展示了本文所研究的应用程序的结果数据集。第4.2节介绍了从上述数据集中对Kotlin应用程序进行分类的试探法，用于响应RQ 1。第4.3节介绍了获取用于响应RQ 2的Kotlin代码比例的方法。第4.4节介绍了源代码的不同发展趋势，用于回答RQ 3。第4.5节描述了参考文献中的代码异味，用于响应RQ 4。最后，第4.6节描述了1)计算Android应用程序质量分数的技术，以及2)测量引入Kotlin代码(用于响应RQ 5)的影响的流程。

### Kotlin应用程序数据集的创建

#### 构建移动应用数据集的标准

由于我们实验的目标是研究Kotlin代码在Android应用程序中的使用和质量，我们决定研究至少有一个版本在2017年或以后发布的移动应用程序。随着Kotlin在2017年被宣布为Android开发的官方语言，在此之前，Android开发者无法从Google获得使用Kotlin语言开发Android应用程序的支持。因此，我们认为，那些最新版本可以追溯到2016年或更早的应用程序无法为我们提供太多关于Kotlin语言在Android领域的使用信息。

此外，我们需要我们的Android应用程序数据集包括，对于每个应用程序: a)它的源代码托管在一个代码库(如GitHub)，和b)发布版本的二进制文件(apk)。

#### 选择Android应用的数据集

我们通过合并三个已经定义的数据集来创建我们的移动应用数据集:F-droid、AndroidTimeMachine (Geiger等人，2018b)和AndroZoo (Allix等人，2016)。据我们所知，这是三个最大的公开可用的Android应用数据集，包含最近发布的应用。

现在让我们来介绍每个数据集，并解释选择它们的原因。

F-droid是一个开源Android应用目录，包含1509个应用。每个移动应用有一个或多个版本，每个版本由一个apk表示。在每个应用程序的主页上，F-Droid提供了下载应用程序最后三个版本的链接，以及包含所有版本列表的另一个页面的链接。

F-droid提供了我们需要的所有信息，即访问代码库和应用程序不同发布版本的apk。然而，与其他数据集相比，应用程序的数量(1509)较低。出于这个原因，我们决定挖掘其他Android数据集，目标是在我们的研究中包括更多的应用程序。

AndroidTimeMachine是一个Android应用程序的图形数据库，可在GitHub和Google Play上访问(Geiger等人，2018b)。为了创建这个数据集，作者定义并执行了一个4步流程:1)识别GitHub上托管的开源Android应用程序，2)提取它们的包名，3)检查它们在Google Play store上的可用性，4)将每个GitHub存储库与其在Google Play store中对应的应用程序条目进行匹配(Geiger et al. 2018b)。AndroidTimeMachine总共有8，431个应用程序，它基于BigQuery中公开提供的GitHub镜像。使用Neo4j数据库可以检索每个应用程序的源代码库链接和应用程序的包名。但是，该数据集不提供来自其应用程序的任何apk。为此，我们决定在AndroZoo数据集上挖掘缺失的apk。

AndroZoo是从各种数据源(Allix et al. 2016)收集的数百万Android应用程序的数据集，包括主要市场Google Play、Anzhi和AppChine，以及较小的目录mobile、AnGeeks、Slideme、ProAndroid、HiApk和F-Droid (Geiger和Malavolta 2018a)。AndroZoo总共有4390288个应用程序，对应总共7795372个APK(版本)。19可用apk的列表在AndroZoo网站上定期更新，同时更新的还有每个应用程序的元数据、主包名称、APK的大小、版本以及应用程序的下载市场等(Allix等人，2016年)。然而，AndroZoo不提供应用程序源代码库的链接，即使应用程序是开源的。

#### 构建我们的研究数据集

现在，我们介绍构建本研究中使用的数据集的不同步骤。

步骤1: 挖掘F-Droid使用F-Droid中包含的上传版本日期，我们从F-Droid中检索到926个符合我们选择标准的应用程序。找到的对应于这些应用程序的版本(即apk)总数为13，094。我们可以下载11，675个apk(89%)。其余的apks文件(11%)不可用。

步骤2: 挖掘AndrodTimeMachine Geiger等人(2018b)展示的AndroidTimeMachine版本包含8431个应用程序，它于2017年10月执行。因为它们的基础设施是公开可用的，我们在2018年10月重新执行了它，目标是纳入更多最近的应用程序。

一旦我们检索到应用程序的列表，不同于Geiger等人(2018b)的原始工作，我们执行了一个新的步骤来保持其代码库只有一个Android清单文件的应用程序。我们添加了这个新步骤，因为我们发现当一个存储库有多个AndroidManifest.xml文件时，AndroidTimeMachine匹配算法会产生错误的结果(即，应用程序与错误的存储库链接)。在我们的附录中，我们列出了遇到上述问题的应用程序。

表1显示了我们从每一步的重新执行中发现的数字与Geiger等人(2018b)的原始数据集的数字之间的比较。总之，我们保留了来自AndroidTimeMachine的2156份申请，它们都符合我们的选择标准。

表1构建AndroidTimeMachine最新版本所执行的步骤

Steps AndroidTimeMachine

|  |  |  |
| --- | --- | --- |
|  | Original | Updated |
| Finding Android Manifest Files | 378,610 | 358,518 |
| Extracting package Names | 112,153 | 114,152 |
| Discarding repositories with more than one manifest file | – | 49,570 |
| Filtering applications on Google Play | 9,478 | 3,664 |
| Matching of Google Play pages to GitHub repositories | 8,431 | 3,664 |
| Filtering applications from 2017 or later | – | 2,156 |

步骤3: 将AndroidTimeMachine与Android zoo相结合:由于AndroidTimeMachine不提供来自应用程序的apk，我们从Android zoo中检索与Android time machine的每个应用程序相对应的apk。为了获得这些apk，我们首先提取位于Android manifest上的包名，然后查询AndroZoo HTTP API。AndroZoo中总共有1531个来自AndrodTimeMachine的应用程序。为了确保这些应用程序与存储库正确链接，我们手动检查了每个应用程序的存储库页面和谷歌Play商店页面。然后，我们发现54个应用程序链接到了错误的存储库。我们删除了它们，剩下1，477个应用程序。

步骤4: 结合F-Droid和AndroZoo最后，我们保留了AndroZoo的1241个应用程序。其余的236个已经在第一步中被F-Droid发现了。

生成的数据集，我们的最终数据集名为Fama zoa(F-droid Android time Machine AndroZoo开源应用程序)，对应于:

Fama zoa = f droid ∨( Android zoo∩Android time machine)

FAMAZOA总共有2，167份申请(926 + 1，241)和19，838份apk。图1显示了每个应用程序的apk(发布版本)的分布。从F-droid我们平均下载6个apk，从AndroidZoo我们平均每个应用程序下载3个apk。总的来说，FAMAZOA每份申请的apk中位数为4。FAMAZOA申请的完整列表在我们的附录中公开提供。

### 检测Kotlin应用程序

为了回答我们的RQ1(在我们的开源应用程序数据集中，Kotlin在移动开发中的采用程度如何？)，我们建立了一个流程来将应用程序和apk(在4.1节中检索)分为三类:1)用Java编写(不包括任何一行Kotlin代码)，2)部分用Kotlin编写，3)完全用Kotlin编写，我们将这些应用程序称为“纯Kotlin”。请注意，我们只关注

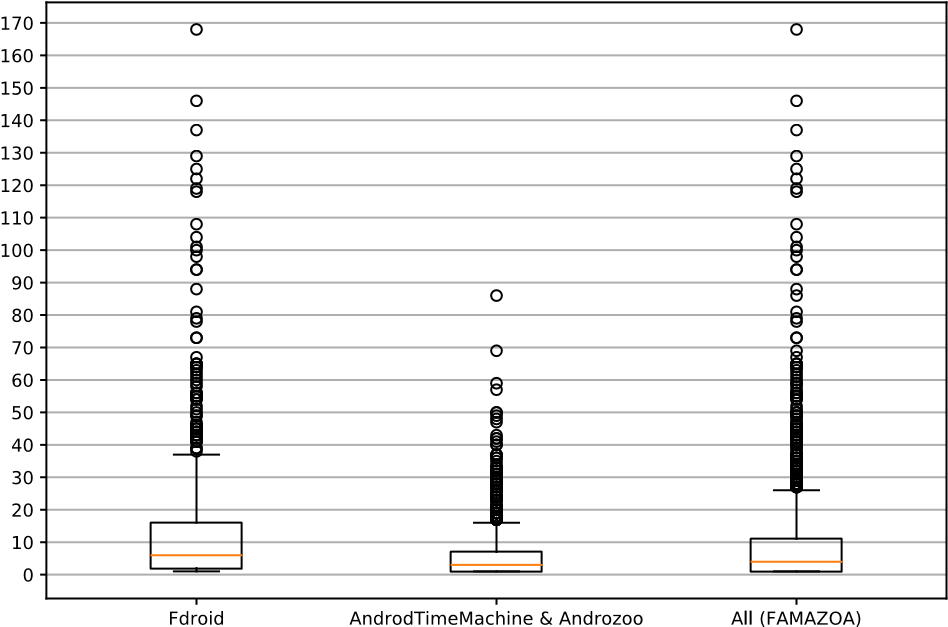


图1每个应用程序的版本号分布

在应用程序的源代码和字节码上，丢弃第三方库，它们的源代码和字节码(jar)都不包含在应用程序的代码库中。

对于FAMAZOA的每个应用程序，它都有一个到代码库的链接和一组apk，我们应用了三种启发式方法对应用程序及其apk进行分类。图2显示了这个分类过程。

我们首先应用了一个启发式Hapk，如图2a所示，它包括在apk文件中查找一个名为kotlin的文件夹。拥有这样的文件夹表明应用程序中存在Kotlin代码。为了自动化这项任务，Android SDK中包含了一个名为apkanalyzer的工具。使用这种启发式方法，我们首先对应用程序的每个版本(apk)进行分类。然后，如果应用程序的至少一个apk被分类为Kotlin，则启发式算法将该应用程序分类为“Kotlin”。否则，它归类为“Java”。Hapk提供了一种廉价而快速的方法来初步猜测Kotlin代码的存在。

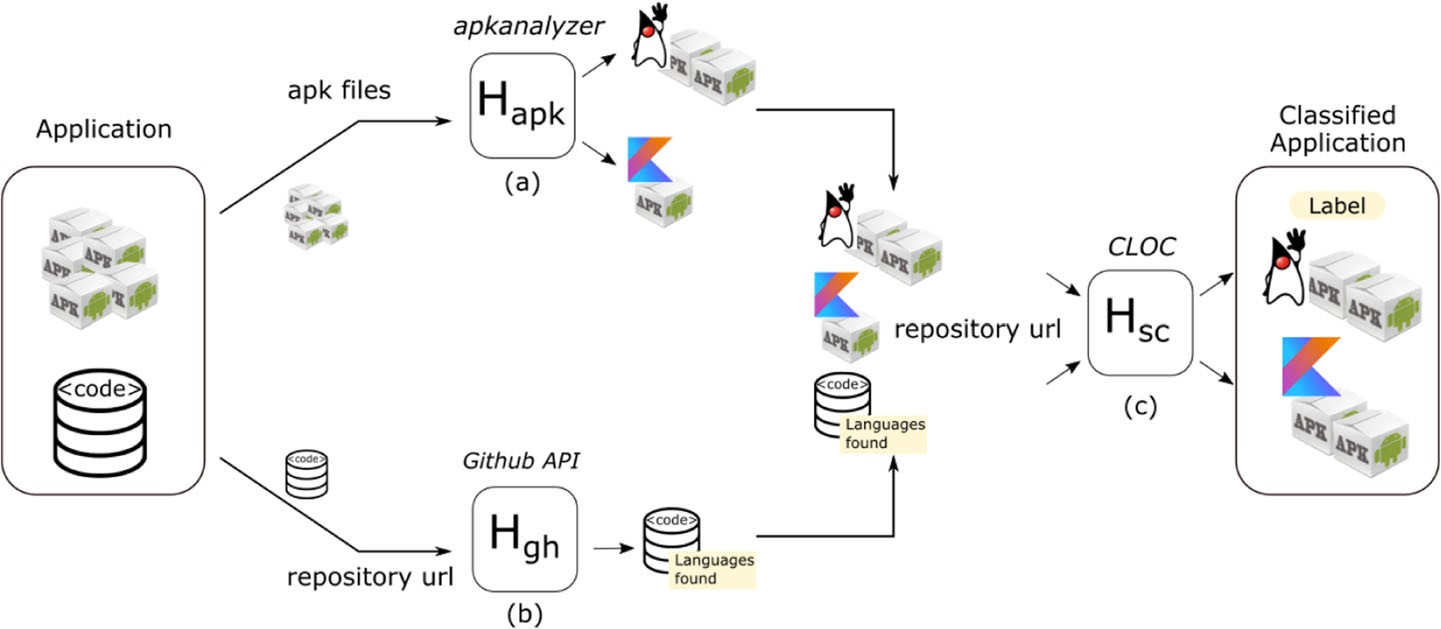


图2我们对Android应用程序进行分类的管道(4.2节。).我们将一个对应于其apks和源代码库url的应用程序作为输入，然后分三步将其分类为Java、Kotlin或Pure Kotlin。首先，在步骤(a)中，启发式Hapk考虑apk的内容。第二，在步骤(b)中，启发式Hgh使用GitHub API来获取关于所使用的编程语言的信息，如果应用程序被托管在那里的话。第三，在步骤(c)中，启发式Hsc分析应用程序的提交历史以验证Kotlin源代码。最后，我们将(标签)分为Java、Kotlin或纯Kotlin

与此同时，我们应用了我们的第二个启发式Hgh，图2b，它依赖于GitHub API:对于GitHub上托管的每个应用程序，我们查询GitHub API以从按编程语言分组的最新版本(即头部)中检索代码量(以字节表示)。如果Kotlin语言出现在API的响应中，我们将应用程序分类为“Kotlin”。这个启发式的输入是一个GitHub存储库的URL。

最后，一旦我们使用Hapk和Hgh检索到一组候选Kotlin应用程序，即Hapk ∪ Hgh，我们就对它们应用启发式Hsc，如图2c所示，以断言Kotlin代码的存在，并测量应用程序具有多少Kotlin。启发式Hsc检查应用程序源代码库的每次提交。如果试探法发现至少一个提交具有Kotlin代码，则应用程序被分类为“Kotlin ”,如果所有提交仅包含Kotlin代码，则被分类为“纯Kotlin”。为了完成这项任务，启发式使用了工具CLOC该函数返回一个列表，其中包含应用程序中使用的语言以及关于非空行的代码量。Hsc非常耗时，因为它需要分析应用程序每次提交的源代码。因此，禁止对我们数据集中的每个应用程序执行Hsc。

注意，与Hsc不同，启发式Hgh只关注GitHub上托管的最新版本(API只检索该信息)。因此，它无法检测以下应用程序:a)不包含最新版本(上次提交)的Kotlin代码，但b)包含旧版本的Kotlin代码。

### 分析Kotlin代码的比例

为了响应RQ2(Kotlin代码在移动应用中的比例是多少？)，我们首先使用4.2节中介绍的试探法检索分类为“Kotlin”或“纯Kotlin”的每个应用程序的最新版本的代码。对于分析中的每个应用程序，我们检索相应的代码库。通过构建，我们的数据集(从FDroid和AndroidTimeMachine数据集构建)只包括代码库公开可用的开源应用程序(第4.1节)。

一旦我们检索了所有的存储库，对于每个存储库(与一个应用程序相关联)，我们对最近的版本(即最后一次提交)执行CLOC，然后我们计算Kotlin代码(不包括空白和注释)占总代码的比例。注意，我们放弃了分析不包含Java或Kotlin代码的文件，比如XML、CSS、JavaScript等。

### 分析Android应用程序的代码演变

为了回应RQ 3(引入Kotlin代码后，代码如何沿着Android应用程序的历史演变？)，我们检查了源代码库，以分析Kotlin代码沿着应用程序历史的演变趋势。我们在本实验中考虑的应用是那些被分类为Kotlin或纯Kotlin的应用(第4.2节)。

对于每个应用程序，我们按照时间顺序(即从最早的提交开始)从其代码库中访问每个提交，以计算与每个提交相关的版本的代码量(也使用CLOC)。

在这个实验中，我们重点分析了两种特定语言的演变趋势:Java(即用于开发Android应用程序的传统语言)和Kotlin。

我们定义了12个代表Kotlin和Java代码不同演化趋势的案例。它们是:

ET1: Kotlin是最初编码的语言，随着历史的发展，Kotlin语言的使用量也在增加 。

ET2: Kotlin代码替换两个连续版本之间的所有Java代码。

ET3: Kotlin代码替换两个连续版本之间的所有Java代码。

ET4: Kotlin与Java一起增长。

ET5: Kotlin增长，Java下降，最新版本的应用程序有两种语言。

ET6: Kotlin增长，Java减少，直到Java代码量为0。

ET7: Kotlin增长，Java保持不变。

ET8: Kotlin不变，Java变。

ET9: Kotlin和Java保持不变。

ET10: Kotlin在应用程序中引入，但最近消失了。

ET11: Java取代了Kotlin代码。

ET12: 其他。

请注意，它可能存在更多的进化趋势，我们没有包括在前面的列表中。我们只包括那些我们在实验中观察到的，并且对本文特别感兴趣的。

为了根据发展趋势对每个Kotlin应用程序进行分类，我们首先绘制了每次提交的Kotlin和Java代码量(代码行数)。图5显示了一些这样的图。然后，我们手动选择最能代表(即，更适合)该应用程序代码演化的演化趋势。只有当论文的两位作者完全同意这种分类时，我们才使用给定的演化趋势对应用程序进行分类。否则，我们将未达成共识的申请归类为其他。此外，我们在附录中公开了所有的图和结果分类，以便进一步分析我们的实验。

### 分析Kotlin和Java应用程序之间的差异以及Android应用程序的质量

对于RQ 4(关于研究的代码异味，Kotlin和Java Android应用程序在代码异味存在方面有区别吗？)，我们在Android应用程序上运行一个代码异味检测工具，该工具来自4.2节中给出的数据集。在本文中，我们主要关注两种异味:面向对象和Android相关代码的异味。

#### 选择代码异味检测工具

为了检测Android应用程序的代码异味，我们选择了名为Paprika的工具(Hecht等人，2015a，b)。选择辣椒粉的原因是: a)它可以检测面向对象和Android特定的代码异味；b)它是为检测Android应用程序上的代码异味而设计的，不需要代码源:辣椒粉的输入是一个Android应用程序的apkc)由于它工作在JVM字节码级别(即，apk包含字节码)，它可以分析用Java和/或Kotlin编写的Android应用程序；d)它是开源的，托管在GitHub上；e)之前的工作广泛使用Paprika来分析手机应用。

**表2** Paprika 可检测的代码异味

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Code smell name | Entity | Considered |
| Object-Oriented | Blob Class (BLOB) | Classes |  |
|  | Swiss Army Knife (SAK) | Interface |  |
|  | Complex Class (CC) | Classes |  |
|  | Long Method (LM) | Method |  |
| Android-Specific | Hashmap Usage (HMU) | Class | X |
|  | Unsupported Hardware Acceleration (UHA) | Class | X |
|  | Leaking Inner Class (LIC) | Inner classes | X |
|  | Member Ignoring Method (MIM) | Methods | X |
|  | Internal Getter/Setter (IGS) | Methods | X |
|  | No Low Memory Resolver (NLMR) | Activities |  |
|  | Heavy ASynctask (HAS) | Async Tasks |  |
|  | Heavy Service Start (HSS) | Services |  |
|  | Heavy Broadcast Receiver (HBR) | Broadcast Receivers |  |
|  | Init OnDraw (IOD) | View |  |
|  | Invalidate Without Rect (IWR) | View | X |
|  | UI Overdraw (UIO) | View |  |
|  | Bitmap Format Usage (BFU) | - | X |

The column ‘Considered’ shows the 10 code smells studied in our work () and the 7 not studied (X)

赫克特等人，2015年a，b，2016年b；Habchi等人2017；Carette等人2017，Grano等人2017；f)工具实施经过了深入验证，包括与Android开发人员共同完成的一项实验(Hecht 2016a)。

表2显示了辣椒粉能够识别的代码异味:4种是面向对象的，13种是Android代码异味。它还展示了与每种代码异味相关的实体。有些实体与面向对象的异味有关(类、方法)，有些与Android异味有关(活动、异步任务)。

辣椒粉的输入是apk(即Android应用程序的一个版本),并返回一个列表，其中包含在该apk中找到的所有异味实例。此外，对于一些异味(包括BLOB，CC和HSS)的每个实例，辣椒粉输出一个使用模糊逻辑计算的模糊值(在0和1之间)(扎德拉1974)。模糊值代表检测实例的真实程度(Habchi et al . 2017赫克特2016a)。

为了检测代码异味的出现，Paprika使用与实体相关的指标。例如，代码异味长方法(LM)是一种面向对象的异味，它与方法有关:LM的一个实例是一种指令数量高于给定阈值的方法。

作为输出，Paprika产生a)发现的异味列表，和b)与用于检测异味的实体相关的度量，例如，方法、活动、服务的数量。那些度量稍后被用来计算软件质量分数(第4.6节)。

#### 我们研究中考虑的代码异味

我们现在描述我们在本文中研究的代码异味。表2在“已考虑”一栏中显示了它们。

首先，我们考虑了四种面向对象的异味(BLOB、SAK和CC，与类相关，LM与方法相关)，因为它们也可以存在于Kotlin应用程序中。让我们简单描述一下它们。BLOB类(Blob)，也称为God类，是一个具有大量属性和/或方法的类(Brown et al. 1998)。瑞士军刀(SAK)是具有大量方法的接口(Hecht 2016a)。复杂类(CC)是包含复杂方法的类。这些类难以理解和维护，需要重构(Fowler et al. 1999)。在辣椒粉上，通过对内部方法复杂性求和来计算类复杂性，并且使用McCabe的圈复杂性来计算方法的复杂性(McCabe 1976Hecht等人，2015年a)。长方法(LM)比其他方法有更多的行，变得复杂，难以理解和维护。

其次，我们考虑了辣椒粉可以检测到的6种Android平台相关代码异味，我们丢弃了7种。我们考虑的那些Android异味是:1) NLMR(与活动相关)，2) HAS(异步taks)，3) HSS(异步taks)，4) HBR(广播接收器)，5) UIO(视图)，以及6) IOD(视图)

让我们简单描述一下它们。当活动没有覆盖方法onLowMemory()时，不会出现低内存解析器(NLMR) (Hecht等人，2015b)，如果活动没有实现该方法，Android系统可能会终止与该活动相关的进程以释放内存，因此，可能会导致程序异常终止(Reimann等人，2014a)。

繁重的异步任务(HAS) (Hecht 2016a)、繁重的服务启动(HSS) (Hecht 2016a)和繁重的广播接收(HBR) (Hecht等人2015a)是相似的:当繁重的操作分别在不同的Android组件、异步任务、服务和广播接收的主线程中执行时，它们就会发生(Mariotti 2013a，b，c)。

UI Overdraw (UIO) (Hecht等人2015a)和Init OnDraw (IOD) (Hecht 2016a)与自定义视图相关。由于缺少方法调用，如clipRect和quickReject (McAnlis 2015)，smell UIO会产生过度绘制视图，这可以避免过度绘制。当在onDraw方法内创建新对象时，IOD发生，该方法可以每秒执行多次，从而导致新对象的多次分配(Ni-Lewis 2015a)。

#### 在我们的研究中忽略了代码异味

我们忽略了辣椒粉可以识别的7种Android相关代码异味。表2在“考虑”一栏中用“X”表示。这些异味是MIM，LIC，IGS，BUF，HMU，UHA和IWR。在本节的剩余部分，我们将描述它们，并解释为什么我们决定将它们排除在外。

当一个方法不访问任何类的属性时，就会出现成员忽略方法(MIM) (Reimann et al. 2014a)。在Android中，建议使用静态方法，因为静态方法调用比动态调用快15%-20%(Android doc 2018 b)。当应用程序使用非静态和匿名的内部类时，会发生异味泄漏内部类(LIC) (Hecht等人，2015b)，因为在Java中，这种类型的内部类包含对外部类的引用，因此它可能会在Android系统中引发内存泄漏(Reimann等人，2014a洛克伍德2013)。我们决定放弃MIM和LIC，因为Kotlin没有静态方法(KotlinDoc 2018)。

内部Getter/Setter (IGS) (Reimann等人，2014a)对应用性能和能耗的影响(Hecht等人，2016b莫拉莱斯等人2016，2017；凯森蒂尼和欧尼2017；格拉诺等人2017；帕隆巴等人2017；Carette等人2017)。然而，这种代码异味只在应用程序运行在Android平台2.3或更低版本时才会产生影响(Cruz和Abreu 2018)。我们摒弃了这种异味，因为运行这些平台版本的活跃Android设备的数量不到0.5%。

位图格式的使用(BFU)与图像格式有关(Carette et al. 2017)。我们抛弃它是因为它既不与Kotlin也不与Java代码相关，也就是说，异味与所使用的编程语言无关。

HashMap使用(HMU) (Carette et al. 2017)发生在开发人员使用小型HashMap实例而不是使用ArrayMap或SimpleArrayMap时，这两者都是由Android框架提供的(Haase 2015AndroidDoc 2018a)。然而,( Saborido et al. 2018)发现的结果显示，就能耗而言，ArrayMap通常比HashMap更慢且效率更低。此外，他们表明，当使用的键是基本类型时，开发人员应该采用稀疏数组变体，因为它们在CPU时间、内存和能耗方面更有效。我们放弃了这种异味，因为:a)提到的(Saborido等人，2018年)的发现，以及b)用于识别HMU事件的辣椒粉机制没有考虑钥匙的类型。

最后，我们丢弃了与自定义视图相关的两种异味。不支持的硬件加速(UHA) (Hecht 2016a)发生在开发人员调用非硬件加速的方法时，因此它在CPU而不是GPU上运行，影响性能和能耗(Ni-Lewis 2015b)。我们抛弃了它，因为这种异味的出现既不依赖于开发人员也不依赖于编程语言。当onDraw方法没有正确实施时，会出现“没有Rect的异味无效(IWR)”(Hecht 2016 a)，从而导致过度绘制视图(Ni-Lewis 2015b)。当开发人员没有指定应该更新的矩形区域时，整个视图都会被重绘，甚至某些区域是不可见的，从而导致性能问题。Ni-Lewis (2015b)指出，开发人员应该调用invalidate(Rect dirty)方法，指定要绘制的区域，以避免这种异味。然而，这个方法在API 28中被否决了，从API 21开始，它的调用被忽略了。因此，我们丢弃了这种异味。

#### 分析Kotlin和Java应用程序在4种面向对象和6种Android代码异味方面的差异

首先，我们分析了FAMAZOA的所有应用程序，以确定4.5.2节中列出的面向对象和Android异味。我们为每个应用程序的所有版本(即apks)运行辣椒粉。然后，我们计算了受每种代码异味影响的Java和纯Kotlin应用程序的百分比。如果应用程序a至少有一个s的实例，则a会受到代码s的影响。这种方法将应用程序分为两组:a)受s影响的应用程序和不受s影响的应用程序。

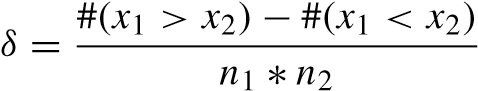
我们还计算了一个指标，该指标计算了一种代码异味的实例数量和与这种异味相关的相关实体数量之间的比率。这一指标的目标是量化异味比例差异的重要性，如Habchi等人(2017年)所做的那样。因此，对于每个应用a，异味s的比率(Habchi等人

2017)定义为:

其中，模糊值(a)是应用程序a中检测到的异味s实例的模糊值(从0到1不等)的总和，实体数(a)是应用程序a中异味s所涉及的实体数。第4.5.1节描述了异味和它们所涉及的实体之间的关系。

我们还计算了Cliff的δ (Romano等人，2006年),它表明了因变量处理的效应大小(Cliff，2014年)的大小。在我们的研究中，我们使用Cliff的δ来确定每种异味s的哪组应用程序(纯Kotlin或Java应用程序)受s影响的实体更多，以及发现的影响大小是否存在显著差异。

Cliff的Delta估计量可以通过(2)获得。

(2)

在该表达式中，x1和x2是组1 (Kotlin应用程序)和组2 (Java应用程序)内的分数，n1和n2是样本组的大小。基数符号#表示计数。该统计估计从其中一组中选择的值(在我们的情况下为a比率)大于从另一组中选择的值的概率，减去反向概率(Macbeth等人，2011年)。

根据Romano等人(2006年)的研究，当0.147 ≤ d < 0.33时，影响大小较小；当0.33 ≤ d < 0.474时，影响大小中等；当d ≥ 0.474时，影响大小较大。我们选择了Cliff的δ检验，因为它适用于非正态分布。而且Cliff的δ也推荐用于不同大小样本的比较(Macbeth et al. 2011)。

### 计算Android应用程序的质量分数

作为对上一个RQ 5的回应(Kotlin的引入对Android应用程序版本质量的积极影响有多频繁？)，我们使用Hecht等人(2015a)提出的技术对移动应用程序的每个版本(apk)进行评分。该分数用于评估特定版本(apk)中的移动应用程序质量，并基于应用程序大小和检测到的代码异味数量之间的一致性。

#### 定义质量模型

为了根据一种类型的代码异味s计算软件质量分数，Hecht等人(2015b)的技术首先使用线性回归建立一个估计模型，该模型表示类型s的代码异味的数量和一个度量之间的关系，该度量根据与s相关的实体的数量表示应用程序的大小。例如，对于异味BLOB，我们考虑的大小度量是类的总数，而对于异味LM，度量是方法的数量。表2展示了异味类型和实体之间的关系。我们为我们在这项研究中考虑的每一种代码异味建立了一个质量模型(4.5.2节)。

为了获得一个应用程序(apk)的质量分数，该技术将代码异味的数量和该应用程序的大小值作为输入，并产生一个分数作为输出。

较高的正分数意味着较好的质量。如Hecht等人(2015a)所述，特定版本的应用程序的软件质量分数计算为残差的加性倒数。因此，较大的正剩余值暗示最差的软件质量，因为它意味着apk相对于其大小具有比标准更多的异味(即，线性回归)，而较大的负剩余值暗示较好的质量，因为异味的数量较少。

#### 培训质量模型

我们为我们在4.5.2节中考虑的每个代码异味创建了一个质量分数模型，即线性回归。我们使用Hecht等人(2015a)定义的数据集训练线性模型，该数据集包含2013年6月至2014年6月期间从Google Play商店提取的3，568个Android版本(apk)。我们选择这个数据集来训练模型有两个主要原因:1)它以前在训练质量模型的类似实验中使用过Hecht等人2015a，2016a以及2)其应用不包括Kotlin代码。从该数据集创建的训练模型代表了在Kotlin发布之前构建的应用程序的质量。因此，我们用它作为基线来衡量Kotlin应用程序是否比Kotlin发布前编写的应用程序质量更好(或更差)。

为了创建质量模型，我们首先在前面提到的数据集上运行辣椒粉。辣椒粉的输出就是训练集。训练数据集的每个元素(一行)对应于单个apk a，并且具有以下信息:a)应用a中异味s的实例数量，以及b)a中与异味s的实体相关联的值。例如，对于异味长方法(LM ),我们计算以下各项之间的线性回归:1)异味LM的实例数量，以及2)方法的总数。

#### 使用质量模型测量Kotlin应用程序的质量

训练完成后，我们从分类为“Kotlin”(第4.2节)的应用程序数据集中计算每个apk的质量分数(每个代码异味一个)。这些apk符合我们的测试数据集。注意，训练数据集不包括来自测试数据集的任何apk。因此，我们排除了模型过度拟合的可能性。

#### 衡量对引入Kotlin质量的影响

我们测量了在一个应用程序中引入Kotlin代码对质量的影响，如下所示。我们研究的应用程序是那些最初有一个或多个分类为Java的apk，然后有一个或多个分类为Kotlin的apk的应用程序，使用4.2节中介绍的启发式方法。对于这些应用程序中的每一个以及每一个代码异味s，我们首先比较了引入Kotlin代码的apk和以前的apk(即，有Java代码但没有Kotlin)之间的质量分数。然后，我们比较了上一个Java apk(即，在引入Kotlin代码之前的版本)和最新的可用Kotlin版本之间的质量分数。这两个比较有不同的目标:第一个旨在衡量在一个应用程序中引入Kotlin后的影响；第二个目标是研究应用程序(现在包括Kotlin代码)发展后的影响。

|  |  |  |  |
| --- | --- | --- | --- |
| Information | Total | Kotlin | Java |
| Unique apps | 2,167 | 244 | 1923 |
| Versions | 19,838 | 1590 | 18248 |

**Table 3** 根据编程语言对 FAMAZOA 的应用程序进行分类

#### 检测引入Kotlin后质量演变趋势的变化

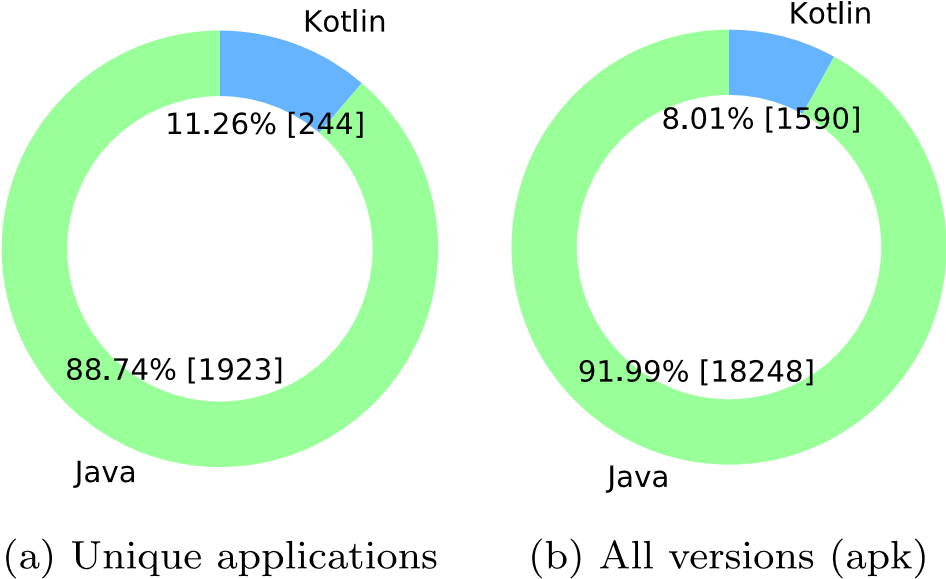
质量演进趋势描述了应用程序版本的质量分数如何随着其演进而变化。在本文中，我们研究了在应用程序a中引入Kotlin代码是否会对质量发展趋势产生积极的影响。我们分析了Hecht等人(2015a)定义的5个主要质量演变趋势:A)持续下降，B)持续上升，C)稳定，D)突然下降，E)突然上升。

对于每个应用A，我们对A上引入Kotlin之前和之后的质量演变趋势进行手动分类。然后，我们认为Kotlin的引入产生了积极的变化，如果:1)引入之前的趋势是“下降”或“稳定”(趋势A、C或D)；以及2)引入之后的趋势仅仅是“上升”(趋势B或E)。请注意，我们放弃了对那些趋势(之前或之后)不符合任何已定义趋势的应用程序的分析。

## 结果

### RQ1:在我们的开源应用程序数据集中，Kotlin在移动开发中的采用程度如何？

表3总结了使用第4.2节中介绍的方法对应用程序进行的分类。FAMAZOA有2，167个应用程序，使用我们的试探法，我们将244个(11.26%)应用程序归类为“Kotlin”应用程序。因此，剩下的1，923个(88.74%)应用程序被归类为“Java”。图3a显示了这些百分比。考虑到版本数量(apk)，我们发现有Kotlin代码的apk有1590个(8.01%)，没有Kotlin代码的有18248个(91.98%)。



图Kotlin和Java应用程序和版本之间的分布

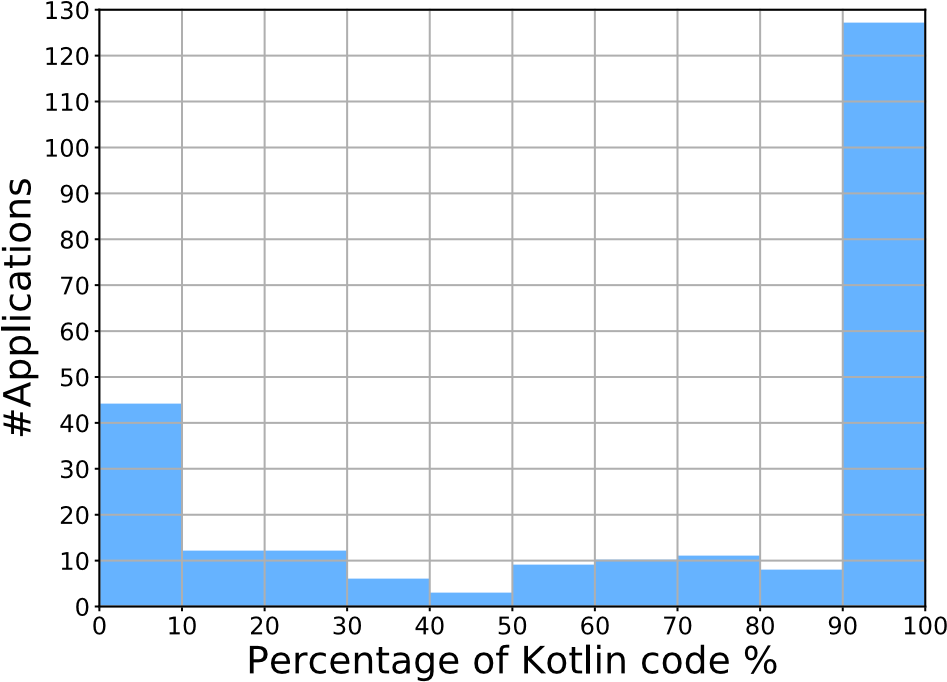
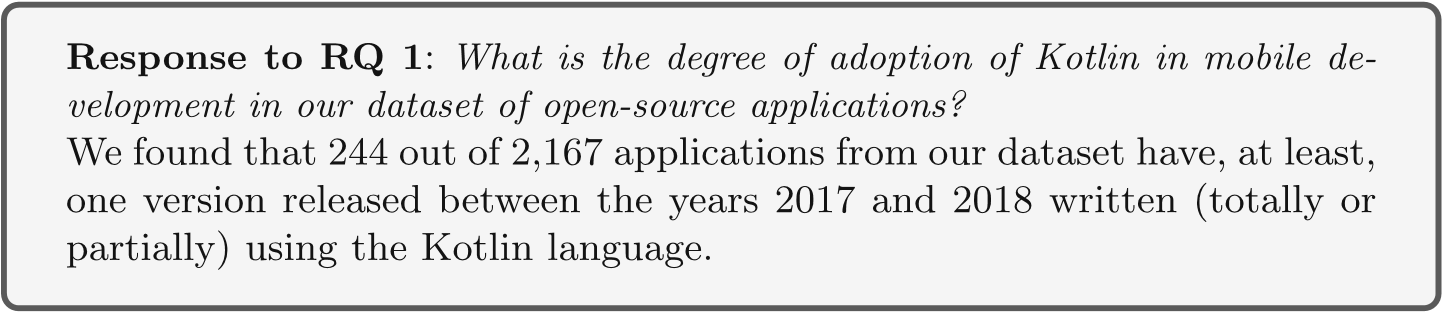


图4根据Kotlin代码百分比的应用程序分布。59.43%的Kotlin应用程序有超过80%的源代码是用Kotlin编写的

现在，让我们解释一下我们是如何检测到来自FAMAZOA的244个Kotlin应用程序的。首先，启发式Hapk (apk分析器，第4.2节)将265个应用程序分类为“Kotlin”。这些应用程序至少有一个分类为“Kotlin”的apk。其中76个国家的所有apk都被归类为“Kotlin”。

然后，Hgh (GitHub API，第4.2节)将234个应用程序归类为“Kotlin”。其中193种也被Hapk归类为“Kotlin”。到目前为止，这两种启发式算法已经将297个独特的应用程序归类为“Kotlin”。

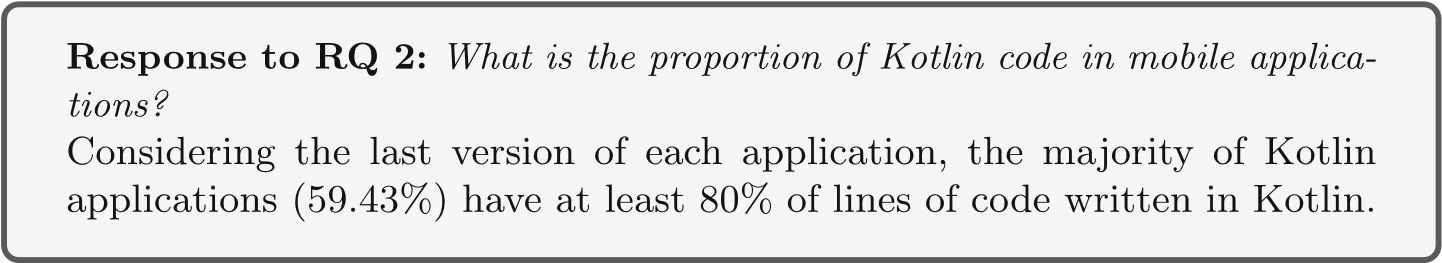
最后，我们对这297个应用程序的存储库应用了Hsc(源代码分析第4.2节)，发现有244个包含Kotlin代码。Kotlin应用程序列表可以在我们的附录中找到。



### RQ2。Kotlin代码在移动应用中的占比是多少？

为了计算Kotlin代码的百分比，我们对至少包含一个Kotlin代码提交的2244个应用程序执行了4.3节中介绍的方法。

图4根据Kotlin代码的百分比显示了Kotlin应用程序的分布。我们发现244个应用程序中有82个(33.61%)只有Kotlin代码。其余的Kotlin应用程序(66.39%)也是用Java编写的。此外，我们发现244个应用程序中有145个(59.43%)具有至少80%的Kotlin代码，相反，2167个应用程序中有45个(18.44%)具有不到10%的Kotlin代码。



### **RQ3: 引入Kotlin代码后，代码如何沿着一个Android应用程序的历史演变？**

我们根据4.4节中介绍的代码发展趋势对每个Kotlin应用程序进行了分类。表4示出了结果，图5示出了对于每个代码演进趋势，作为示例的一个特定应用的代码演进。

我们发现最常见的代码演变趋势是ET 5，在244个Kotlin应用程序中有52个(21.3%)。这种发展趋势代表了这样的情况，在引入Kotlin代码的第一个版本(即commit)之后，Kotlin代码的数量趋于增长，而Java代码的数量减少。图5e显示了app JalkametriAndroid的代码演变，这与演变趋势相对应。尽管如此，JalkametriAndroid的最新版本比Kotlin有更多的Java代码行。另一个归类为ET 5的应用是Poet-Assistant(图5f)。但是，与Jalkametri-Android不同的是，上一版本的Kotlin代码量大于Java代码量。在提到的应用程序中，代表Kotlin代码演变的线条似乎相对于Java是对称的。我们假设在这些情况下，用Java代码编写的应用程序的一些组件逐渐迁移到了Kotlin代码。

与ET 5相似的趋势是ET 6: 48个Android应用程序(19.7%)显示了这一趋势。与ET 5不同，trend ET 6中的Java代码量逐渐减少，直到达到

表4根据Kotlin和Java源代码的演变趋势对Android应用的分类

|  |  |  |  |
| --- | --- | --- | --- |
| Source code evolution trend | | # Apps | % |
| ET 1 | Kotlin is the initial language and the amount of Kotlin grows | 19 | 7.8 |
| ET 2 | Kotlin code replaces all Java code | 15 | 6.1 |
| ET 3 | Kotlin code replaces some Java then Java continues growing | 4 | 1.6 |
| ET 4 | Kotlin increase together with Java | 8 | 3.3 |
| ET 5 | Kotlin grows and Java decreases (but never is zero) | 52 | 21.3 |
| ET 6 | Kotlin grows and Java decreases until the Java code is 0 | 48 | 19.7 |
| ET 7 | Kotlin grows and Java remains constant | 41 | 16.8 |
| ET 8 | Kotlin is constant and Java changes | 43 | 17.6 |
| ET 9 | Kotlin and Java remain constant | 7 | 2.9 |
| ET 10 | Kotlin introduced but lately disappears | 3 | 1.2 |
| ET 11 | Java replaces Kotlin code | 2 | 0.8 |
| ET 12 | Other | 2 | 0.8 |
| Total applications | | 244 | 100% |

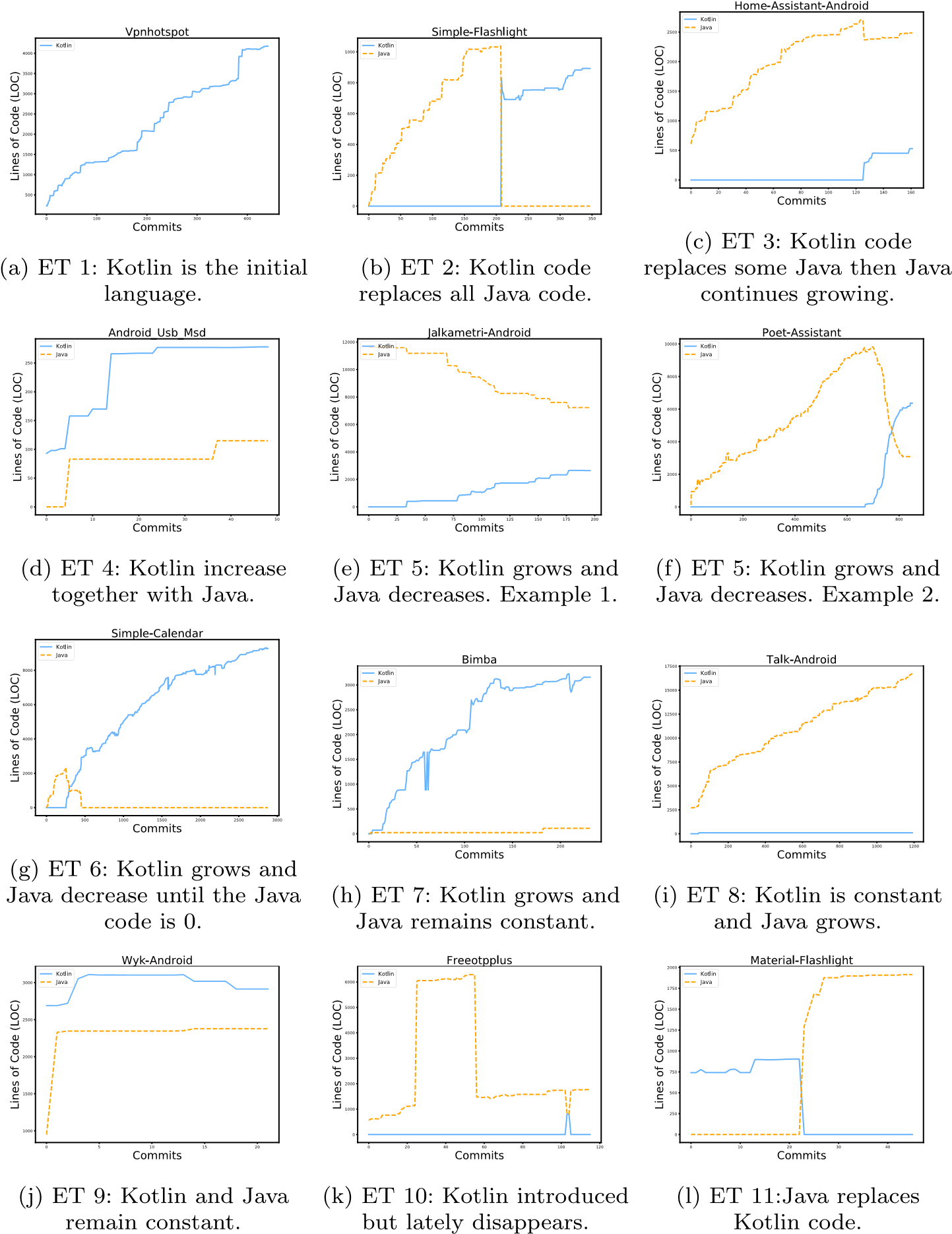


图5 kot Lin和Java代码的演化趋势。该图展示了4.4节中描述的每个演进趋势的一个图形。每个图形展示了Kotlin和Java代码沿着单个应用程序的历史(即提交)的演进。x轴对应于提交，y轴对应于代码量(即行数)

零锁定。从那时起，这些应用程序不再包含Java代码。图5g显示了一个应用程序Simple-Calendar，其第一个版本是用Java编写的。然后，从commit 09ef99到206dfe的版本，引入了Kotlin代码，去掉了Java代码。最后，从提交eee184开始，应用程序仅由Kotlin代码组成。

有两个趋势ET 7和8(分别有41和43个应用程序。)的代码量几乎是不变的。例如，图5e显示了分类为ET 8的Talk-Android的代码演变。一次提交(7f12)引入了一部分Kotlin代码(105行)。从那以后:a)kot Lin代码的数量在演进中保持不变(最后一次提交724有106个LOC)，Java代码的数量不断增长。图5h显示了相反的情况(app Bimba):Java代码的数量是恒定的，而Kotlin代码的数量在增长。还有7个应用程序，其用两种语言编写的代码量保持不变，例如图5j中所示的应用程序Wyk-Android。这些应用代表了发展趋势。

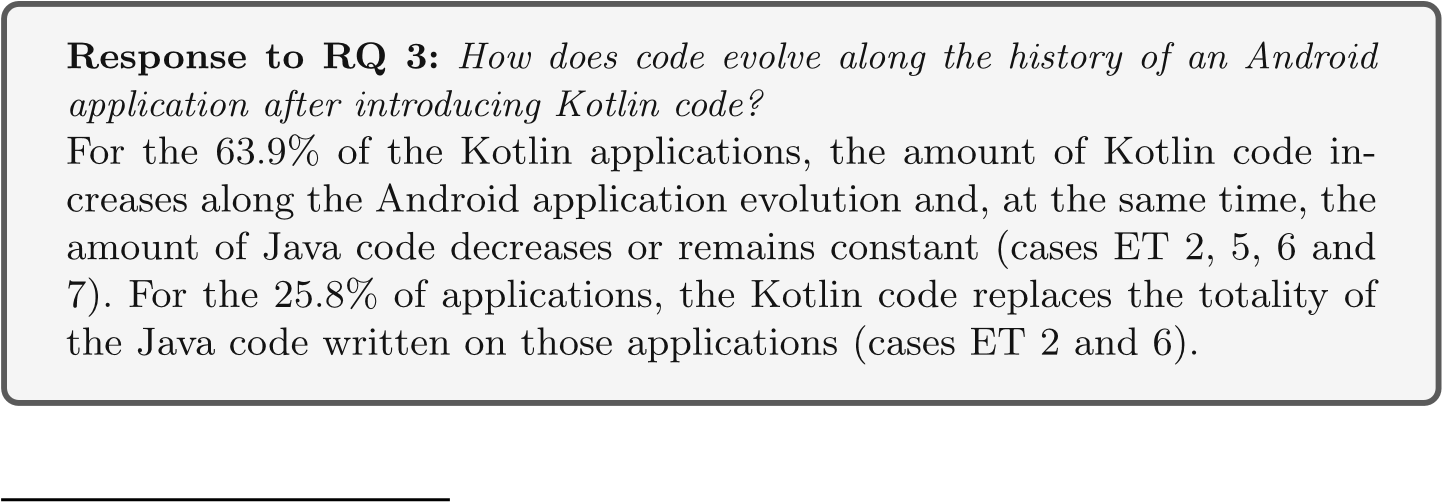
第五个最频繁的演进趋势是ET 1，有19个应用程序:这些应用程序最初是用Kotlin编写的，不包含任何版本的Java代码。应用程序Vpnhotspot就是其中之一(图5a)。evolution trend ET 2，有15个应用程序超过244个(6.1%)，代表了图5b所示的简单闪存应用程序，这些应用程序最初是用Java编写的，然后在一次提交(18b5c9)中，将完整的代码库从Java迁移到Kotlin。然而，与ET 5和6不同，在ET 2中没有版本共享Java和Kotlin代码。另一方面，我们用趋势ET 11对2个应用进行分类，即，当与ET 2比较时，其发展呈现相反行为的应用，例如应用材料-手电筒，图5l。这些应用程序从Kotlin迁移到Java代码。

此外，有8个Android应用程序(3.3%)对应于trend ET 4:Java和Kotlin的数量都在增长。图5d示出了应用Android USB MSD的代码量:在那里，用一种语言编写的代码的引入没有产生用另一种语言编写的代码量的减少。

另一个演进趋势是ET 3:当Kotlin代码被引入时，Java代码的数量减少(以相似的比例),但随后Java的数量又开始增长。图5c显示了应用程序Home-Assistant的代码演变，这是归类为ET 3的4个应用程序之一(1.6%)。这里，我们怀疑开发人员只迁移了一部分代码。

此外，ET 10代表了3个应用程序:在某个时候引入了Kotlin代码，但在后来的某个版本中，该代码被删除了。图5k显示了Freeotpplus应用程序的代码演变。在这里我们可以看到Kotlin是在commit 2dbc32中引入的，但是在两次提交之后，它被完全移除了。

最后，我们将trend ET 12 (Other)分配给2个应用程序，我们不能分配任何我们先前定义的演进趋势。在我们的附录中，我们介绍了本节中讨论的进化趋势的分类。



### RQ4: 关于所研究的代码气味，Kotlin 和 Java Android 应用程序在代码气味存在方面是否存在差异？

我们应用了第4.5节中介绍的方法。我们对所有来自FAMAZOA的apk执行辣椒粉。Paprika成功分析了来自2，040个应用程序的17，725个apk(94%)，并对来自127个应用程序的所有apk抛出了一个错误。

为了回答这个研究问题，我们比较了Java应用程序和纯Kotlin应用程序的结果。首先，我们着重计算受影响的应用程序的数量。接下来，我们关注受影响实体的数量。

#### 受影响的应用程序数量

表5示出了对于每种编程语言和代码异味，受这种代码异味影响的Android应用的百分比，即，具有一个或多个异味实例。

我们发现四分之三(75%)的面向对象的异味影响了超过93%的应用程序，包括Kotlin和Java两种语言。LM是最常见的异味，影响两种语言中大约99%的应用程序。SAK是最不常见的异味，但它仍然影响大多数应用程序，大约65%的Kotlin应用程序和66%的Java应用程序。因此，我们的结果与之前的工作一致(Hecht等人，2015aHabchi et al. 2017)显示LM、CC、BLOB和SAK分别是Android应用中最常见的OO异味。

表5还显示了用Java和Kotlin编写的应用程序的百分比差异(K - J行):对于所有OO异味(LM、CC、BLOB和SAK)，差异很小:分别为0.18%、1.61%、1.59%和-0.84%。我们的结论是双重的。首先，四分之三(75%)的面向对象代码异味更多地出现在Kotlin应用程序中。然而，它们对Kotlin和Java应用程序的影响比例相似。

关于Android异味，我们观察到NLMR是最常见的异味，影响了99%的Kotlin和98%的Java应用程序。此外，第二个最常见的Android异味UIO，它分别影响54%和45%的Kotlin和Java应用程序。第三个最常见的异味是HBR，它影响了35%的Kotlin和39%的Java应用程序。其他3种异味(HSS、HAS、IOD)最多出现在22%的应用中。请注意，有3种Android异味会相应地影响更多的Java应用程序。而且，应用程序受Android异味影响的比例最显著的差异是13.06%来自HAS异味。

我们观察到大量应用程序受到NLMR异味的影响，因为它与Android应用程序的主要组件Activities有关，几乎每个Android应用程序中都存在这种异味。另一方面，我们发现较少数量的应用程序受到剩余异味的影响。由于HBR、HSS和HAS与其他Android组件BroadcastReceiver、Service和AsyncTask相关，这些组件对于所有Android应用程序来说并不重要，并且由于UIO和IOD与视图组件相关，因此仅影响自定义视图的实现。尽管如此，由于Android SDK提供了几种视图实现，大多数应用程序不需要自定义视图实现。

**表5**

percentageofandroidapplicationaffectedbycodesmell

朗格

%Affectedapplicationsbysmells

面向对象的技能

男性异味

longcomplexsblobswissnolowui

HeavyHeavyHeavyInit

方法classclassarmymemoryoverdrawbroadcastserviceasynctaskondraw

刀具溶解器

接收器启动

LM)(CC)(BLOB)(SAK)(NLMR)(UIO)(HBR)(HSS)(HAS)(IOD)

)

（

purekotlin 99 . 8098 . 5095 . 1265 . 5555555656

09.55

爪哇

99.6296.8993.5366.5198.8445.3339.9819.9222.61

06.50

千焦

0.181.611.59

−

0.840.468.69

−

4.26

−

2.61

−

13.063.05

阿纳普

a

isaffectbedbyacodesmeet

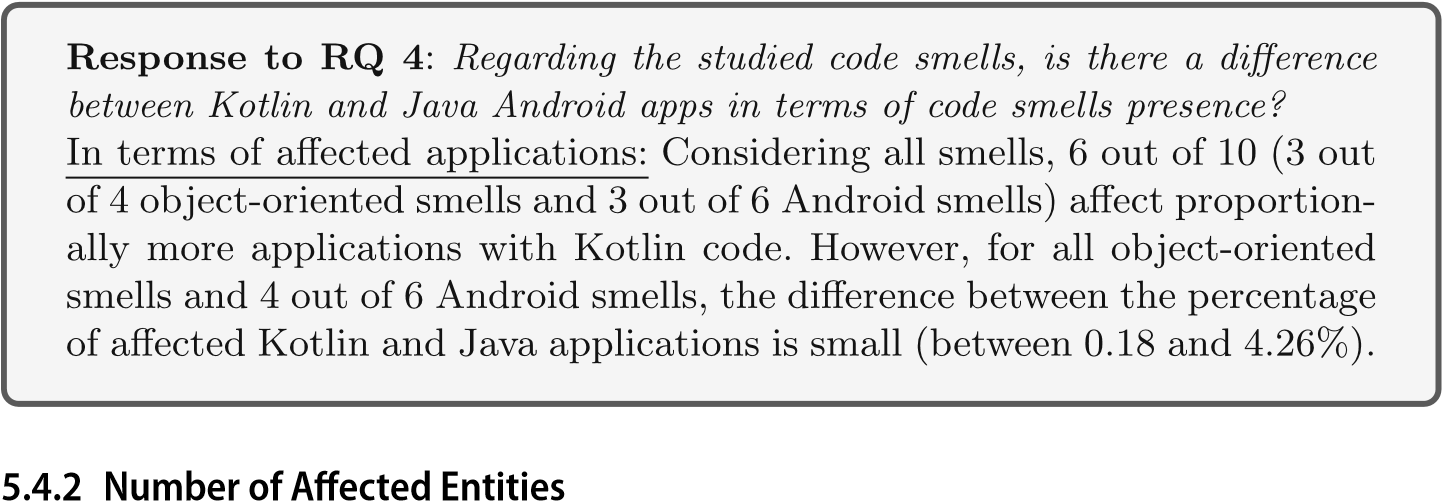
s

如果

a

hasatleastoneinstanceof

s



#### 受影响的实体数量

现在，我们使用4.5.4节中介绍的方法来研究受异味影响的实体的比例。表6显示了对于每种异味和编程语言，应用程序中异味比率的中值(med)(公式(1))和Cliff的δ效应大小。

首先，我们观察到最常见的异味(LM、CC和BLOB)具有较小的中值比率。这意味着，尽管它们存在于大多数应用中，但只有少数实体会受到这些异味的影响。

此外，Cliff的δ值表明，纯Kotlin和Java应用程序的异味中值比率之间的差异对于所有面向对象的异味来说都具有统计学意义:CC和SAK的差异“很小”，LM和BLOB的差异“中等”。我们的结论是，尽管4个面向对象的异味中有3个会影响更多的Kotlin应用程序，但我们的结果显示

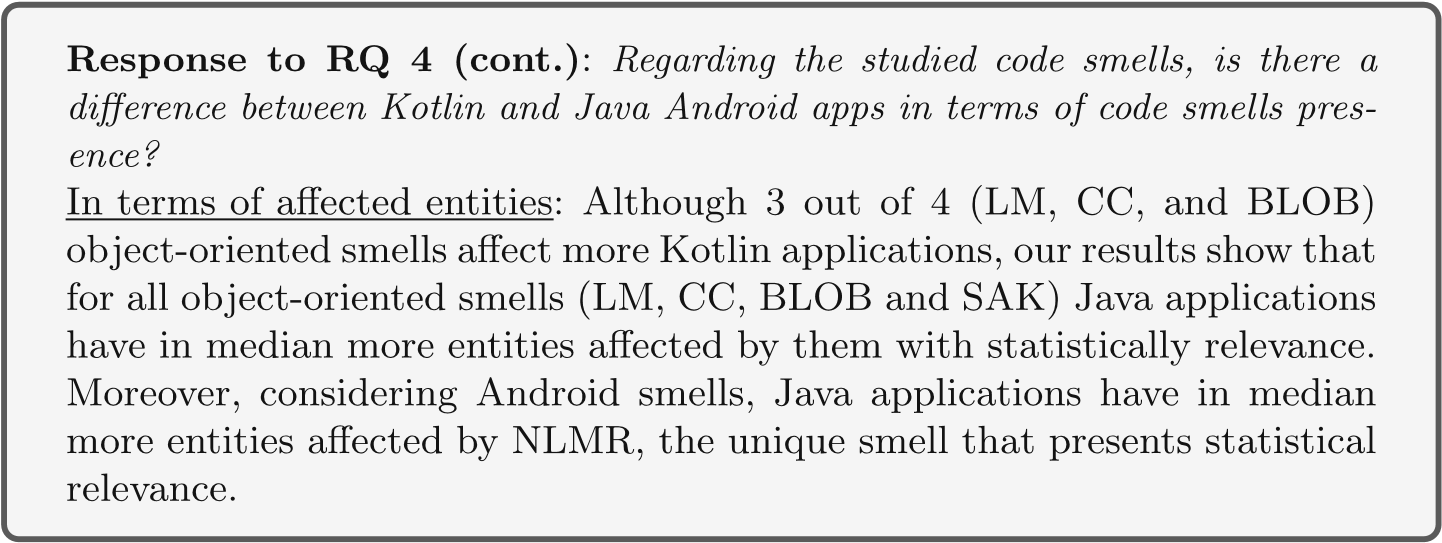
表6 otlin和Java的比率比较

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Smell | Lang | Median | Cliff’s δ | Significance of |
|  |  | Ratio |  | difference |
| LM | Kotlin | 0.0563 | −0.3873 | Medium |
|  | Java | 0.0736 |  |  |
| CC | Kotlin | 0.0593 | −0.3168 | Small |
|  | Java | 0.0781 |  |  |
| BLOB | Kotlin | 0.0163 | −0.4338 | Medium |
|  | Java | 0.0278 |  |  |
| SAK | Kotlin | 0.0008 | −0.2433 | Small |
|  | Java | 0.0040 |  |  |
| NLMR | Kotlin | 0.3750 | −0.2915 | Small |
|  | Java | 1.0000 |  |  |
| UIO | Kotlin | 0.0769 | 0.1156 | Insignificant |
|  | Java | 0.0000 |  |  |
| HBR | Kotlin | 0.0000 | −0.0699 | Insignificant |
|  | Java | 0.0000 |  |  |
| HSS | Kotlin | 0.0000 | −0.0240 | Insignificant |
|  | Java | 0.0000 |  |  |
| HAS | Kotlin | 0.0000 | −0.1306 | Insignificant |
|  | Java | 0.0000 |  |  |
| IOD | Kotlin | 0.0000 | 0.0341 | Insignificant |
|  | Java | 0.0000 |  |  |

“Cliff的δ”列显示了纯Kotlin和Java应用程序的异味中值比率之间的差异:负值意味着Kotlin中的异味影响的实体比Java中的少

对于100%的面向对象的异味，Java应用程序中有更多的实体受到它们的影响，具有统计相关性。

关于Android异味，我们观察到，对于Java和Kotlin应用程序，6种异味中的4种(HBR、HSS、HAS和IOD)的中值比率为零，这与表5一致，因为这两种语言的应用程序受这些异味影响的比例不到50%。此外，Cliff的δ显示6种Android异味中的5种没有显著差异，包括前面提到的那些。我们的结论是，很少有实体会受到这些异味的影响，即使HBR影响了超过35%的Android应用程序(见表5)。关于异味NLMR，我们观察到，具有统计显著性的是，Java应用程序受影响的实体更多，而差异“很小”。这一结果与Habchi等人(2017)的结果一致，并表明NLMR影响大多数Java应用程序的所有活动。



### RQ5: Kotlin的引入对Android应用程序版本质量的积极影响有多频繁？

使用辣椒粉的输出来计算得分质量，辣椒粉的输出作为输入apks。在我们的实验中，我们发现总共有57个应用程序，最初只有Java代码，后来引入了Kotlin。考虑到这些应用，表7显示了在引入Kotlin代码后，每种异味的质量得分增加的Kotlin应用数量。与一种异味相关联的一个质量分数的增加意味着该异味的实例更少，因此应用的质量更好。

结果显示，对于10种异味，至少50%的引入了Kotlin代码的应用程序，它们的质量分数在最后一个Java版本和引入Kotlin的第一个版本之间增加了(见表4“第一个Kotlin”列)。这意味着，对于这样的应用程序，Kotlin代码的引入对质量分数产生了积极的影响。请注意，50%的值是下限值，因为异味LM精确地改善了50%的应用。然而，所有其他异味改善50%以上。例如，表7显示，对于10种异味中的8种，质量改进应用的百分比大于62%，对于4种异味大于76%。这意味着，仅仅对于一种异味(LM ), 50%的应用程序并没有提高质量。对于其他异味，有所改善的应用数量大于没有改善的应用数量。此外，对于所有异味，至少50%的应用程序在最新Java版本和最新(即最新)Kotlin版本之间的质量得分有所提高(见表4“最新Kotlin”列)。

表7 引入Kotlin后质量分数的变化

Code smell # Apps Kotlin improves quality score Positive change

on evolution trend

First Kotlin

Last Kotlin

|  |  |  |  |
| --- | --- | --- | --- |
| LM | 28(50%) | 28(50%) | 10/28(35.71%) |
| CC | 35(62.5%) | 36(64.29%) | 13/36(36.11%) |
| BLOB | 43(76.79%) | 42(75%) | 18/42(42.86%) |
| SAK | 44(78.57%) | 45(80.36%) | 17/45(37.78%) |
| HBR | 45(80.36%) | 40(71.43%) | 7/40(17.5%) |
| HAS | 37(66.07%) | 30(53.57%) | 2/30(6.67%) |
| HSS | 45(80.36%) | 42(75%) | 6/42(14.29%) |
| IOD | 36(64.29%) | 36(64.29%) | 5/36(13.89%) |
| NLMR | 33(58.93%) | 31(55.36%) | 6/31(19.35%) |
| UIO | 36(64.29%) | 36(64.29%) | 8/36(22.22%) |

该表显示了提高了与异味相关的质量分数的应用数量。“第一个Kotlin”列显示了没有Kotlin的最后一个版本和有Kotlin的第一个版本的质量分数之间的比较。“最后一个Kotlin”列比较了没有Kotlin的最后一个版本和有Kotlin的最后一个版本的质量分数。“进化趋势的积极变化”一栏统计了Kotlin的引入导致质量进化趋势从“下降”或“稳定”变为“上升”的应用数量

例如，让我们关注表7第二行的异味CC(复杂类)。如“第一个Kotlin”栏所示，对于56个应用中的35个(62.5%)，引入Kotlin代码的版本vk比没有kot Lin vk 1的上一版本(即vk的前一版本)具有更大(即更好)的与异味CC相关的质量分数。图6a示出了与“Mozilla Klar”应用的每个版本的异味CC相关联的质量分数。不包含Kotlin代码的最后一个版本对应于图中的X=5。我们观察到具有Kotlin代码(X=6)的第一个版本增加了质量分数，以及所有后续版本。

此外，如列“最后的Kotlin”所示，对于36个应用(64.29%)，具有Kotlin代码的最新版本具有比引入Kotlin之前的版本更大(即更好)的与异味CC相关联的质量分数。同样，“Mozilla Klar”就是那些应用中的一个:上一个版本(X=12)比上一个Java版本(X=5)得分高。注意，对于CC异味，有一个应用程序(36 - 35)的质量分数:a)在其引入Kotlin的版本中下降，但是b)在最近的Kotlin版本中上升。在图6b中显示了被命名为“HTTP快捷方式”的那些应用之一的质量分数的演变。

我们还在表7中观察到，对于10种异味中的5种，在第一个Kotlin版本后质量提高的应用数量(“第一个Kotlin”列)大于在最后一个Kotlin版本后质量提高的应用数量(“最后一个Kotlin”列)。这意味着一些Kotlin应用程序的质量分数在第一个和最后一个Kotlin版本之间会下降。例如，图6c示出了“Hanks Note”应用的质量分数的演变:第一个Kotlin版本相对于最后一个Java版本增加了分数。然而，在随后的版本中，质量分数下降，甚至低于上一个Java版本的分数。最后，图6d示出了演变

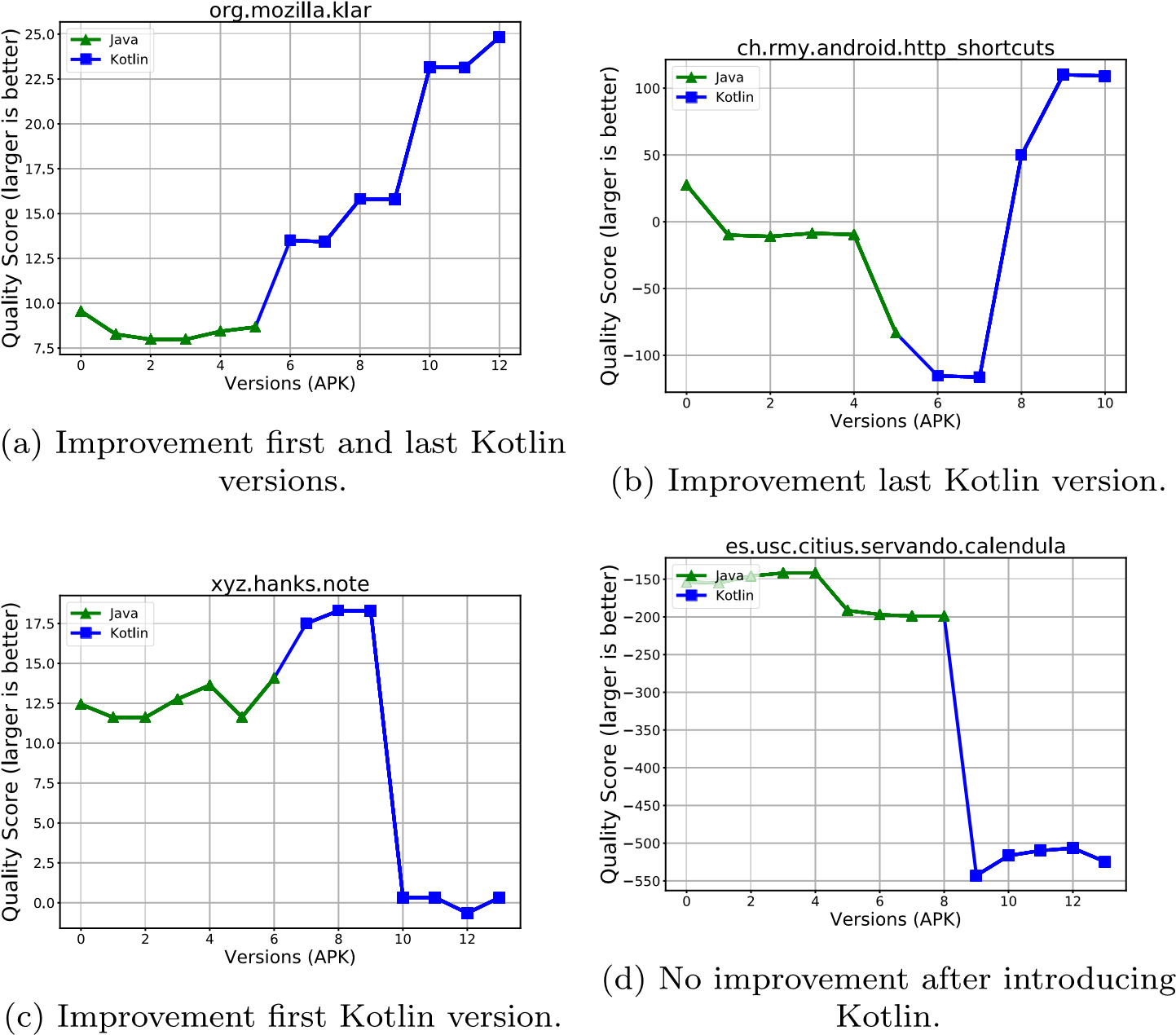
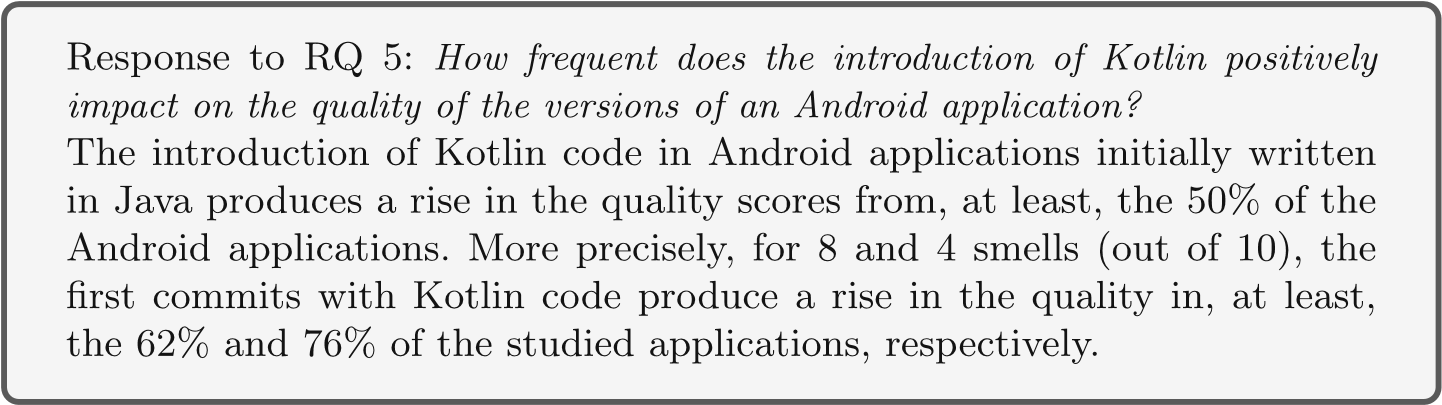


图6基于CC异味的质量分数沿版本历史的演变

一个名为“Calendula”的应用程序的质量分数，在引入Kotlin代码后，它的质量分数不断下降。这一发现表明，Kotlin应用程序的质量也会随着应用程序的发展而下降。



最后，表7的最后一列显示了Kotlin的引入改变了质量演变趋势，从“下降”或“稳定”到“上升”的应用数量(第4.6.5节)。我们称这些积极的变化为质量进化变化。对于面向对象的异味，在Java的最后一个版本和Kotlin的最后一个版本之间出现质量改进的应用程序的百分比在35.7%和

42.8%.对于Android异味，积极改变的应用数量较低:在6.6%到22.2%之间。

图7显示了三种情况。图7a中显示的第一个对应于质量演变趋势的积极变化。在推出科特林之前，质量分不断下滑。Kotlin的引入积极地改变了进化趋势:此后，质量分数不断上升。第二种情况，图7b，显示了Kotlin的引入没有改变趋势:引入前后的质量分数是稳定的。注意，在进化结束时，质量分数突然上升。然而，我们没有将这种上升与Kotlin的引入联系起来，kot Lin的引入在以前已经做了很多。最后，第三种情况，图7c，没有呈现趋势上的变化:质量分数在引入Kotlin之前上升，并且在那之后继续上升。

总之，关于质量演变趋势变化的研究表明，一些应用程序:a)用Java编写的应用程序的质量不再持续下降，b)在引入Kotlin代码后，质量甚至有所提高。

## 对有效性的威胁

### 内部

Android应用程序的分类在第4.2节中，我们定义了一个基于三种检查源代码和apk的试探法将Android应用程序分类为“Kotlin”和“Java”的过程。通过应用这些试探法，我们确保在我们的数据集中没有假阳性，即被分类为“Kotlin”但在其生命周期中没有Kotlin代码的应用程序。然而，它可能存在一些假阴性，即，应用程序:1)具有Kotlin的版本，但是启发式Hapk不将它们分类为Kotlin，以及2)来自代码储存库的最后版本不再具有Kotlin代码(根据定义，未被启发式Hgh检测到)。在Hapk失败的情况下，启发式Hsc可以检测到这样的应用程序，但是考虑到我们当前的基础设施，在完整的FAMAZOA数据集上执行Hsc是昂贵的(耗时的)(请记住，Hsc从应用程序的源代码中分析每个提交)。

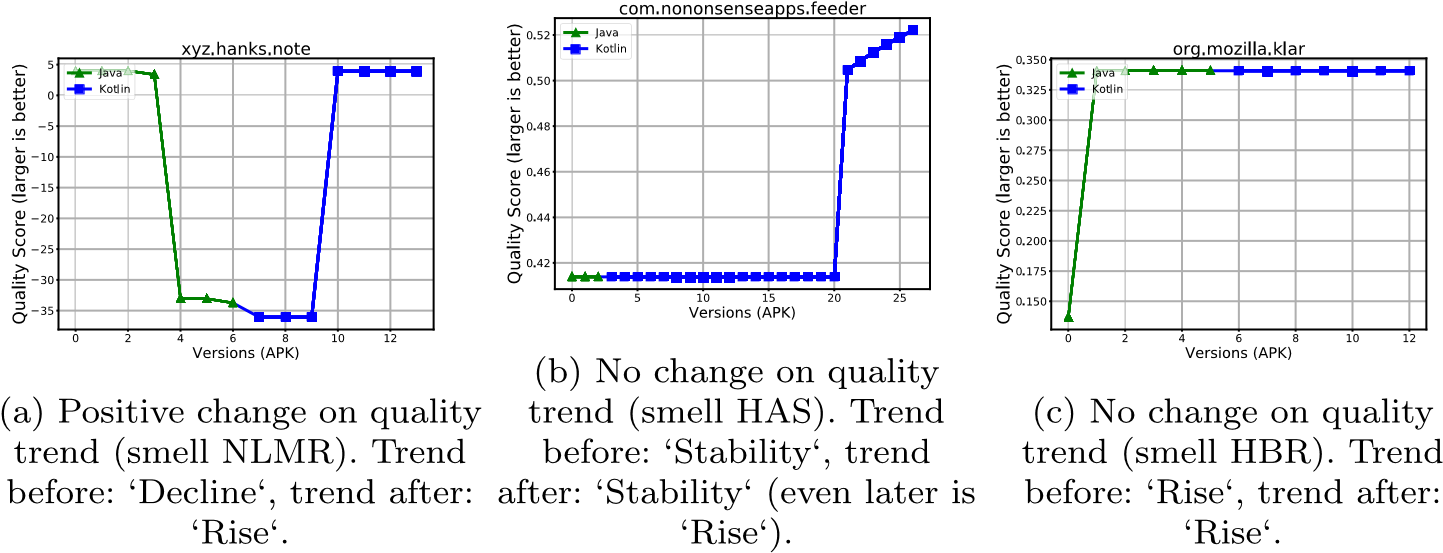


图7引入Kotlin代码前后质量趋势演变分类的三种情况

软件质量模型的模型化存在训练数据集不具有代表性的风险。因此，质量模型产生不正确的估计。我们考虑了以前的工作用于创建质量模型和检测Android应用程序上的异味的相同训练数据集(Hecht等人，2015a，b，2016b)。此外，该数据集的使用允许训练数据集和验证数据集没有任何交集，从而避免生成过度拟合的模型。

为了公平地比较Java和Kotlin应用程序中代码异味的存在，我们选择了Paprika tool，它在字节码级别工作。然而，我们的工作中没有考虑到其他的代码异味，这可能会影响代码的总体质量。尽管如此，根据Mannan等人(2016年)的说法，在这项工作中考虑的4个面向对象代码异味中有3个在Android应用程序中被研究最多的代码异味的排名中。

此外，Kotlin提供了新的特性和不同的语法，可以引入新类型的代码异味。然而，据我们所知，文献中没有研究调查采用Kotlin的影响，以及其特性对源代码质量的影响。

### 外部

**Paprika的有效性**：Paprika可能存在以下风险:a)假阳性，即它检测到不正确的异味实例，以及b)假阴性，即它检测不到异味实例。然而，辣椒粉已经通过不同的实验进行了详尽的评估(Hecht，2016a)。

**FAMAZO的代表性：**我们的实验集中在研究移动应用程序的源代码和字节码。出于这个原因，我们决定研究GitHub上的开源应用程序。据我们所知，结合F-Droid、AndroidTimeMachine和AndroZoo，使我们的数据集成为最大的Android存储库，其中包含每个应用程序的二进制代码和源代码。然而，我们不能把我们的发现推广到非开源的应用上。

**缺失版本(apk)**：存在一个风险，即F-droid和AndroZoo不包含Android应用程序的所有已发布版本。因此，这些丢失的数据可能会影响我们对应用程序质量的分析，这是基于对F-droid所有可用apk的分析。

**Kotlin和Java代码的比较**：为了比较每个应用程序上Kotlin和Java代码的数量，我们使用了工具CLOC。因此，我们的结果依赖于该工具的精确性。工具上的一个错误可能会影响应用程序生命周期中两种语言的演化分析。

**开发者背景和团队组建：**在我们的工作中，我们分析了不同Android应用的源代码。然而，我们没有考虑关于开发人员的信息，因为使用特定语言的经验，也没有考虑关于开发人员团队的信息。有可能更有经验的开发人员会比新手写出更好的代码。然而，我们分析的所有应用程序都是F-droid或Google Play发布的，因此它代表了Android开发的当前快照。

## 讨论

自2017年5月17日谷歌I/O 2017期间宣布Kotlin为官方Android编程语言以来，移动开发者已经可以使用一种新的现代JVM语言来开发Android应用程序。这种语言有新的特性，比如lambdas(不可用于使用Java开发Android应用程序),它旨在促进开发任务。在本文中，我们首先着重研究这种新语言的采用程度。我们发现大约11.26%的开源应用程序使用过Kotlin代码。这一发现表明，除了最近宣布的支持，Kotlin已经获得了非常稳定的Java语言和使用Java创建Android应用程序的开发基础设施的重要份额。此外，我们发现对于一些应用程序，6.1%的开发人员在几个连续的提交中从Java迁移到Kotlin。

文档中提到的Kotlin的优势之一是它与Java的互操作性:开发人员可以使用Java和Kotlin编写应用程序。我们在本文中发现，我们的数据集中有10.38%的应用程序同时使用了Kotlin和Java。

尽管Kotlin开发人员社区声称支持Kotlin，但没有研究采用Kotlin的影响。然后，这项工作旨在通过衡量用Kotlin和Java编写的Android应用程序的质量来支持开发人员的决策。我们发现，在最初用Java编写的应用程序中引入Koltin后，至少有50%的应用程序的质量得分有所提高。由于没有工作比较Kotlin和Java应用程序中代码异味的存在，也没有工作分析在用Java编写的移动Android应用程序中引入Kotlin的影响，我们的结果为移动Android应用程序中代码异味的研究带来了新的数据。

## **结论和未来工作**

在过去的几年中，出现了不同的开发方法来开发移动应用程序。在这种背景下，谷歌已经宣布Kotlin在2017年成为Android开发的官方支持语言。在发布近一年后，我们进行了一项实证研究，以验证使用这种新编程语言开发的Android应用程序是否比使用传统方法(即使用Java语言)编写的应用程序质量更好。

为了实现我们的实证研究，我们首先通过挖掘3个现有的Android数据集(F-Droid、AndroidTimeMachine和AndroZoo)创建了我们的开源Android应用研究数据集，名为FAMAZOA。数据集由以下应用程序组成:a)代码库公开可用，b)存在2017年后发布的版本(apk)。

FAMAZOA总共包含2，167个应用程序和19，838个apk。然后，我们定义并应用了3种试探法，根据Kotlin代码的数量对这些应用程序进行分类。我们的研究发现，2167个应用程序中有244个在2017年至2018年期间至少有一个版本是使用Kotlin编写的。

我们的研究首先集中于分析Kotlin和Java代码数量的演变。我们发现，对于包含Kotlin代码的63.9%的应用程序，用该语言编写的代码量随着应用程序的发展而增加，同时，Java代码量减少或保持不变。此外，对于所研究的25.8%的应用程序，Kotlin完全取代了Java。

然后，我们的研究通过从我们的数据集中分析应用程序中代码异味的存在来关注Android应用程序的质量。我们发现，至少在93%的Java和Kotlin应用程序中都存在四分之三的面向对象(OO)异味(LM、CC和BLOB)。从百分比来看，4 OO种异味中有3种(LM、CC和BLOB)在Kotlin应用中更为常见。然而，我们发现Java应用程序有更多的实体受到10个代码异味中的5个的影响，包括所有OO异味。对于其余的异味，我们发现没有统计学上的差异。最后，我们发现，在用Java编写的Android应用程序中引入Kotlin代码，至少提高了50%的研究应用程序的质量。

作为未来的工作，我们计划从经验上验证kot Lin development community(3.4节)所声称的优势在移动开发环境中是否成立，并调查更多与性能和能耗相关的代码异味和反模式。