arXiv:2304.01102v1 [cs.SE] 3 Apr 2023

RunBugRun

面向自动程序修复的可执行数据集

Julian Aron Prenner
prenner@inf.unibz.it

Romain Robbes
romain.robbes@u-bordeaux.fr

2023年4月4日

摘要

近年来，我们可以观察到自动程序修复（APR）正转向数据驱动技术，尤其是深度神经网络。这意味着需要使用数十万乃至数百万个不可执行代码片段进行训练。我们希望让人们更多关注神经程序修复（NPR）中一个常被忽视的方面，即代码的执行。代码执行有几项显著优势：它支持对候选修复进行基于测试的评估，并能提供有助于修复的宝贵信息。在本研究中，我们推出了一个完全可执行的数据集，其中包含45万个有缺陷/已修复的程序对。这些程序最初发布于编程竞赛网站，使用八种不同的编程语言编写。除数据集外，我们还提供了用于编译、安全执行和测试程序的基础设施，以及细粒度的缺陷类型标签。作为参考，我们提供了两种基线的基本评估结果：一种基于生成并验证的方法，另一种基于深度学习。借助该数据集，我们希望实现多个目标：推动神经程序修复超越完全静态的代码表示，促进对基于执行的特征的使用，并通过纳入多种不同的编程语言，平衡当前自动程序修复数据集和基准测试中 Java 占主导地位的局面。

关键词 自动程序修复、数据驱动的软件工程、故障定位

1 引言

软件缺陷不仅会在美国单独造成数十亿美元的成本 [9, 37]，还需要开发人员投入大量精力 [38]，而且修复周期往往长达数月 [36]。因此，人们对自动发现并修复缺陷抱有浓厚兴趣，这并不令人意外。这正是自动程序修复（APR）的目标。

从 GenProg [23, 24], 开始，所谓的生成并验证（G&V）方法在过去十年中主导了自动程序修复研究。大多数基于 G&V 的系统依赖基于频谱的故障定位，首先确定一组可疑（即很可能存在缺陷）的代码位置，然后自动程序修复工具便会集中精力修复这些位置。基于频谱的故障定位本质上依赖测试执行（因而也依赖程序执行）来确定可能的缺陷位置。同样，测试和程序执行也用于评估：通过所有测试用例是补丁正确性的必要条件（但并非总是充分条件）。因此，在 G&V 中，执行是故障定位和正确性评估的核心环节。有时，程序执行甚至会用于实际的修复过程。例如，DynaMoth [20] 会收集运行时信息，并将其整合到修复策略中。沿着这一思路，重要的自动程序修复基准测试集都是由可执行代码及配套的完整测试套件构成的（见第 2 节和表1）。

近年来，越来越多的自动程序修复研究建立在深度学习之上。这一子领域甚至有了自己的名称：神经程序修复（NPR）。神经程序修复系统使用多达数百万组含缺陷与已修复的代码片段对进行训练。迄今为止，研究重点主要放在静态代码特征上，尤其是文本特征 [13, 47, 32], , 较少关注树或图表示 [43, 17]。

由于 NPR 系统需要大量数据，且大规模手动收集和隔离软件缺陷并不可行，因此人们会从开源代码仓库（例如 GitHub）中挖掘大量缺陷数据 [71]。

```
包 主程序
导入 "fmt"
函数 主程序() {
  // B - ss 的代码
  变量 S字符串
  fmt.Scanf("%s" , &S)
  对于 i := 1;  i < len(S);  i++ {
    if (长度(S)-i)%2  != 0 {
      continue
    } 否则如果 S[0:((len(S)-i)/2)]  == \
                S[((长度(S)-i)/2):长度(S)-i]  {
      fmt.Println(长度(S)  - i)
      跳出
    }
  }
}
```

ID	780269		
语言	Go		
拆分	Test		
标签	控制 −流程中断添加		
失败测试	输入	预期输出 6 ...	实际输出 6\n4\n2 ...
通过测试	输入	预期输出 8 ...	实际输出 8\n ...
错误消息	None		

图1：所呈现数据集中的缺陷实例示例及其元数据。每个缺陷都包含：a) 所用的编程语言，b) 拆分（即训练、验证集或测试），c) 一个或多个层次化缺陷标签，d) 失败和通过的测试用例列表，以及 e) 发生运行时错误或异常时的一条或多条错误消息。更多示例见图12和图 2。

我们希望借助此数据集，将传统自动程序修复基准测试集的特征（可执行性、测试用例）与 NPR 数据集的特征（规模大、训练数据）结合起来。我们的工作基于以下动机：1）需要大规模的可执行代码；2）需要多语言性；3）需要更好地整理大规模数据集，并深入了解其性能（见第3节）。我们在此概述这些动机：

大规模执行的必要性。对于NPR数据集，含缺陷代码通常会从上下文中“撕出”，作为单纯的代码片段加入训练集，并保留缺陷位置上下方各几十行代码。与机器学习中的惯例一样，可用数据随后会被划分为三个（或更多）拆分或集合。通常包括：a) 训练集，b) 用于模型参数调优的验证集，以及 c) 用于最终性能验证的测试集。所有集合都应遵循相同的数据分布。由于原始数据通常由不可执行代码片段组成，验证集和测试集也同样如此。因此，评估必须采用静态指标，即在词法层面将模型的输出与真实值进行比较（例如精确匹配或BLEU）。然而，由于程序语义可以通过许多不同的词法形式表达（见图 2），这类静态或词法指标存在问题，会导致假阴性。另一方面，执行可以让我们对修复候选方案运行测试套件，从而在语义层面评估正确性。

许多神经程序修复系统除了在分布内测试集上进行评估外，也会使用 Defects4J [34] 或 Bugs.jar [64] 等成熟的基于测试的自动程序修复基准测试集进行评估，从而能够与生成与验证系统直接比较。然而，这些基准测试集也存在一些问题。首先，它们由大型 Java 项目组成，编译耗时较长，拖慢了评估过程。由于这类基准测试集中的缺陷通常需要人工整理（例如添加测试用例 [46]），因此这些数据集规模相对较小，尤其是与神经程序修复所使用的大型训练集相比。许多神经程序修复系统通常只在这些基准测试集的子集上进行评估（例如，因为它们仅处理单语句、单行或单代码块缺陷） [88]，这进一步加剧了规模不足的问题。规模较小的基准测试集也可能更容易受到基准测试过拟合问题的影响；此前的研究曾报告过这一问题 [54, 46]。

对多语言自动程序修复的需求我们看到，Java 编程语言在自动程序修复领域占据明显主导地位。尽管目前似乎越来越多的人开始关注其他语言 [16, 76, 26], , 但迄今为止，许多流行语言在自动程序修复领域的代表性仍严重不足（例如 Go、PHP、Ruby）。我们倡导自动程序修复采取更加多语言化的方向。此前研究的发现也支持开发多语言或语言无关的自动程序修复系统。例如，Campos 和 Maia [10] 发现，在所研究的开源 GitHub Java 项目中，44% 使用了不止一种编程语言；48% 的缺陷修复提交修改了非 Java 源文件。

更好地整理数据并获得洞见的必要性。大规模数据集的一个问题是，我们很难了解其中包含哪些内容。这会影响数据质量：大规模数据集中的提交真的是缺陷修复吗？如果是，它们的难度是否适当？是否有某类缺陷修复所占比例过高？第二个问题涉及自动程序修复模型的性能洞见，而这些洞见可以指导未来的改进。某个模型是否在

“简单”缺陷上表现出色，却在更难的缺陷上表现不佳？某个模型是否在某一类缺陷上表现很好，却在其他类别上表现不佳？

RunBugRun。 在这项工作中，我们介绍了 RunBugRun，这是一个用于自动程序修复的新数据集和基准测试集（我们将该数据集的测试集视为基准测试集），旨在解决上述问题：

- 首先，RunBugRun 支持大规模执行：正如我们在第 4 节中详细介绍的，我们从提交至程序设计竞赛的简短算法程序中收集了 450,000 个软件缺陷。所有代码都可以完整编译和执行。最重要的是，我们为每个软件缺陷提供测试用例，并提供类似 Defects4J 的基础设施，用于编译、安全执行（在沙箱中）和测试程序。我们还收集了大量异常堆栈跟踪和错误消息；近期研究表明，这些信息有助于定位并修复软件缺陷 [83]。
- 其次，RunBugRun 收录了以八种不同流行编程语言（C语言、C++、Java、Python、JavaScript、Ruby、Go、PHP）编写的程序，有望为自动程序修复迈向更加多语言化的未来贡献力量。其中四种语言是静态类型语言（C语言、C++、Java、Go），另外四种（Python、JavaScript、Ruby、PHP）是动态类型语言。此外，各语言数据集规模的差异为研究迁移学习方法提供了机会。
- 第三，RunBugRun 的 45 万个软件缺陷来自一个规模大得多、但经过精心筛选的数据集。我们确保纳入的确实是软件缺陷（存在失败的测试用例），并且有恰当的修复方案（修复后测试通过）。我们还确保这些软件缺陷类型恰当（属于语义缺陷而非语法缺陷），且难度分布合理。我们也处理了重复和不稳定性等问题。此外，我们为大多数软件缺陷标注了细粒度类别，便于诊断自动程序修复系统的“弱项”。

在第 4 节中，我们将说明如何构建 RunBugRun。这些软件缺陷提取自 ProjectCodeNet [57], , 这是一个包含大量编程竞赛网站代码提交的数据集。本节将详细介绍我们所采用的大规模筛选、预处理、执行和标注步骤。

第 5 节介绍最终数据集的特征，包括编程语言、修复规模、错误标签、异常与错误以及数据拆分的统计信息。我们还展示了两个自动程序修复基线的初步结果。一个是 Cardumen，即一种生成与验证方法 [52]。另一个是基于微调后的预训练语言模型 CodeT5 [75] 的 NPR 基线。除了直接比较性能外，我们还讨论了基线评估中的其他发现。我们展示了修复性能与修复变更数量之间存在很强的相关性，指出了两个基线各自的优势和弱项，并提供了初步证据，表明神经程序修复模型能够跨语言边界学习和应用修复方案（知识迁移）。

根据这些结果，我们得出结论：G&V和NPR方法都有很大的改进空间。我们将在第6节讨论一些未来的发展方向和面临的挑战。
RunBugRun中的大多数程序都是用于解决特定问题的简短算法实现，因此与大型软件项目有很大不同。我们将在第7节讨论这一点以及其他偏差和局限性。最后，我们将在第8节总结本文。

2 相关工作

我们首先介绍现有的自动程序修复基准和NPR数据集，发现它们呈现出一种两极分化：一方面是规模较小且可执行的自动程序修复基准，另一方面是规模较大但不可执行的NPR训练数据集。表1概述了这些数据集和基准测试，并与我们的方法进行对比。随后，我们进一步介绍从竞赛提交中提取的数据集，因为它们与我们的工作最为相似，并将它们与RunBugRun进行比较。

自动程序修复基准下面简要介绍近期文献中提出的自动程序修复基准。这些基准不提供任何训练数据，通常仅用于评估。所有这些基准都可执行，并附带测试用例。

Defects4J [34]可视为最成功的自动程序修复基准，并已在自动程序修复相关文献中得到广泛使用。它包含从大型且知名的 Java 项目（例如 Apache Commons库）中收集的 365 个（版本 1）和 835 个（版本 2）软件缺陷，这些项目拥有完备的测试套件。Defects4J 提供了基础设施，用于检出项目的缺陷版本或修复版本并运行测试。

表1：自动程序修复基准测试集和数据集概览。只有提供了测试的基准测试集才支持测试驱动评估。其他数据集主要用于数据驱动方法（例如机器学习）

Name	Type	# 软件缺陷	# 项目	编程语言	可执行
Defects4J [34]	真实世界项目	835/395[1]	17/6[1]	Java	✓
Bugs.jar [64]	真实世界项目	1,158	8	Java	✓
Bears [48]	真实世界项目	251	72	Java	✓
QuixBugs [45]	算法实现	40	–	Java、Python	✓
Codeflaws [69]	算法实现	3902	–	C	✓
IntroClass [41], IntroClassJava [21]	学生作业	450/297[2]	–	C语言、Java	✓
BugSwarm [70]	失败和通过的 CI[5]容器	1,940/1,292[6]	–	Java、Python	✓
BugJS [26]	真实世界项目	453	10	JavaScript	✓
BugsInPy [76]	真实世界项目	493	17	Python	✓
Defexts [8]	现实世界的项目	225/301[7]	152/170[7]	Kotlin，Groovy	✓
Ours	算法实现	~450,000	-	C语言、C语言++、Java、Python, JavaScript、Ruby、PHP、Go	✓
ManySStuBs4J [35]	现实世界中的软件缺陷单行补丁（仅差异）	153,652	> 1,000	Java	✗
CodRep [14]	单行补丁	58,069	–	Java	✗
BFP [72]	真实世界缺陷	58,350/65,455[3]	–	Java	✗
CoCoNUT [47]	真实世界缺陷差异和上下文	3,241,966[4]	–	Java、Python JavaScript、C语言	✗
MegaDiff [53]	现实世界中的软件缺陷全文件差异	663,029	–	Java	✗
FixJS [16]	现实世界中的软件缺陷（从 GitHub 提交中挖掘）	323,907	–	JavaScript	✗
TSSB-3M [62]	现实世界中的软件缺陷单语句	3M/9M[8]	–	Python	✗

[1] 分别对应版本 2 和版本 1；[2] 分别对应 C语言 和 Java 版本；[3] 分别对应小型和中型版本；[4] 对应 Java；[5] 持续集成；[6] 分别对应 Java 和 Python（2021年9月）；[7] 分别对应 Kotlin 和 Groovy；[8] 分别对应有和无单语句/多语句修复；

Bugs.jar [64]和 Bears [48]是另外两个包含真实世界 Java 缺陷的自动程序修复基准测试集。与 Defects4J 一样，Bugs.jar 源自规模更大的 Java项目。Bears 中的软件缺陷是通过分析 GitHub 上开源项目失败的 CI 构建发现的。

QuixBugs [45]是一个包含 40 个 Python 和 Java 算法实现的集合（每个实现都有一个存在缺陷的版本和一个修复后的版本）。每个问题都配有若干可通过所提供基础设施运行的测试用例。QuixBugs 中的缺陷相对简单，大多是单行或单个标记缺陷。它只有 40 个缺陷，是规模最小的基准测试集之一；不过，它已被广泛用于自动程序修复领域的研究。

BugsInPy [76]和 Bugs.js [26] 分别构成了 Python 和 JavaScript 中的两个自动程序修复基准测试集。与 Defects4J 一样，它们都包含真实世界项目中的缺陷，并提供测试基础设施。类似地，Defexts [8]是一个面向 Kotlin 和 Groovy 的自动程序修复基准测试集。

BugSwarm [70]是一个集合，包含从开源 Java 和 Python 项目的持续集成流水线中提取的失败与通过的持续集成容器对（类似于 Bears）。它提供了用于下载和运行容器的基础设施。

以上内容主要都是评估基准测试集，因此不提供训练数据。本文的数据集除了包含一个大型训练集外，还包含一个可用作基准测试的测试拆分。该测试拆分的规模远大于现有基准测试（见表1）。

神经程序修复数据集 与基准测试集不同，以下缺陷数据集不包含测试用例。缺陷通常以不可执行代码片段的形式呈现，也就是说，要么是含缺陷代码行及其周围的几行代码，要么是包含缺陷的函数。所提及的数据集主要用于训练和评估神经程序修复模型。ManySStuBs4J 收集了来自 1,000 多个 Java项目的 150,000 多个“愚蠢”缺陷（例如，参数交换、二元运算符错误）。每个缺陷都根据缺陷分类体系进行分类。在 CodeRep 机器学习竞赛中，Chen 和 Monperrus [14] 从开源项目中收集了 58,069 个单行缺陷。Tufano 等人 [72]构建了一个大型数据集，其中包含从 GitHub 挖掘的 Java 方法含缺陷/修复代码对，并用其训练神经网络。类似地，Lutellier 等人 [47] 从四种不同的编程语言中挖掘出 300 多万个含缺陷/修复代码片段对，并用其训练他们的 NPR 模型。Monperrus 等人 [53]发布了 MegaDiff，这是一个包含 600,000 多个 Java 补丁的数据集，补丁中修改行数从 1 行到 40 行不等。Csuvik 和 Vidács [16] 推出了 FixJS，这是一个包含 300,000 对来自 GitHub 的 JavaScript 函数（含缺陷/修复）的数据集。最后，Richter 和 Wehrheim [62] 发布了 TSSB，这是一个大型数据集，包含 900 万对 Python 单语句已修复/含缺陷代码片段。值得注意的是，TSSB-3 是其中的一个子集，包含 300 万对代码片段，其中所有缺陷都可以通过单语句修改修复。

基于竞赛提交的研究与本文最相关的是 Majd 等人的工作，他们通过挖掘 Codeforces 上的提交，收集了一个大型自动程序修复数据集。我们的数据集基于 Project CodeNet，其中包含来自 AIZU 和 AtCoder 的提交。尤其是在数据收集方面，两者存在相似之处。不过，我们似乎采用了更为复杂的筛选和后处理流程。例如，我们运行了每项提交并在所有测试用例上执行，以筛除那些看似正确、实际上却未通过测试的提交。在他们的工作中，只执行了一小部分提交。此外，我们检查了所有提交是否存在语法错误，因为我们只关注语义错误。因此，我们数据集中的所有提交（包括有缺陷的提交）都可以编译。对于 Python，我们还将所有 Python 2 代码转换为 Python 3 代码。在少数情况下，我们甚至处理了 Python 小版本之间的弃用变更（例如，在 Python 3.9 中，最大公约数函数从 fractions 包移到了 math 包）。我们使用基于三元组的度量，确保不同集合之间没有重复项，而他们似乎采用了精确匹配。他们的数据集似乎只提供了非常粗略的缺陷分类（即添加、删除或修改），而我们为每个缺陷提供细粒度层级标签。我们还为简单的 NPR 系统和 G&V 系统提供了性能基线。最后，也是最重要的一点，我们提供了类似 Defects4J 的基础设施，用于编译、测试和执行所有缺陷。

Haque 等人 [27]比较了 NPR 的基于执行和基于匹配的评估指标。在他们的研究中，他们从 Project CodeNet 收集了一个有缺陷/已修复的程序对数据集，本研究也使用了该数据集。我们再次发现，双方在数据收集方式上有许多共同之处。虽然我们只关注逻辑错误，但他们也收集了包含语法错误的提交。与包含八种编程语言的数据集不同，作者仅收集了 Python 和 Java 的提交。Haque 等人的工作似乎主要侧重于比较不同的评估指标，而非构建数据集或基准测试集。因此，他们似乎既未提供通用基础设施，也未提供缺陷分类。

Tan 等人 [69]介绍了 Codeflaws，这是一个基于 Codeforce 提交构建的自动程序修复基准测试集。该基准测试集仅包含 C 语言程序，且不包含训练数据。它包含 7,436 个软件缺陷，也远小于 RunBugRun（测试集中有 30,000 多个软件缺陷）。值得注意的是，Codeflaws 将每个缺陷归入由 40 种不同缺陷类别组成的分类体系。

3 动机

本节将详细说明我们为何认为有必要为自动程序修复创建一个新的数据集。这一观点基于三项主要动因：1）需要大规模执行数据集及其影响（第3节 3.1），2）需要多语言自动程序修复数据集（第3节 3.2），以及 3）需要进行整理和标注（第3节 3.3）。

3.1 自动程序修复需要大规模且可执行数据集

自动程序修复领域由两个子群体组成，它们对数据的不同需求使彼此渐行渐远。一方面，经典的 G&V 方法依赖可执行数据集；由于需要人工评估且执行时间较长，这些可执行数据集难以扩展，并且面临基准测试过拟合的风险。另一方面，NPR 方法使用大规模数据集进行训练。这类数据集不可执行，因此模型无法从程序执行中学习有价值的信息，评估时也只能依赖词法信息（可执行基准测试集除外）；此外，数据规模大且缺乏执行过程，也会导致一些数据质量问题。
因此，我们认为，由大量小型、可执行且经过充分测试的程序构成的数据集，是一个很好的折中方案。这样的数据集具有可扩展性，因为运行时评估的成本并不高。扩大规模可以增加软件缺陷类型的多样性，降低基准测试过拟合的风险，并提供人工评估之外的替代方案。最后，训练期间能够获取执行数据，也为 NPR 从新的信息来源中学习提供了潜力。我们将在以下段落中详细阐述这些观点。

让 NPR 与 G&V 更加接近 如前所述，大多数成熟的自动程序修复基准测试集都是在生成与验证方法处于研究核心地位的时期创建的。因此，这些基准测试集满足了 G&V 系统的需求：它们完全可执行，从而能够进行基于频谱的故障定位以及运行测试以开展评估。收集并整理大量高质量的软件缺陷非常耗时。因此，这些基准测试集按照当今的标准规模相对较小，也就不足为奇了。
另一方面，日益流行的基于神经网络的方法（NPR）是在大量数据上训练的。按照机器学习中的惯例，会留出一定比例的数据（通常为 5%–15%），用于验证（例如，调整超参数）和评估模型。NPR 借鉴了许多神经自然语言处理（NLP）技术，常常采用相同的模型架构和分词算法。在许多情况下，NPR 也遵循 NLP 的评估实践，使用 BLEU、ROUGE 或 n-gram 相似度等静态指标。然而，与自然语言不同，代码是可执行的。通过提供规模相对较大的完全可执行数据集，NPR 与传统自动程序修复方法可以采用共同的评估规范，从而便于直接比较性能。

基准测试过拟合问题 由于现有自动程序修复基准测试集没有提供训练数据，基准测试集中的缺陷可能会被用于调优和优化自动程序修复系统（例如，重点关注这些基准测试集中出现的缺陷模式）。这可能导致系统偏向特定基准测试集。以往研究发现，不同基准测试集之间，以及真实世界缺陷与基准测试集中的缺陷之间，都存在显著的性能差距（基准测试过拟合）。例如，Noda 等人 [54] 发现，与在常见基准测试集上的表现相比，Elixir [65] 在工业环境中的表现较差。同样，Liu 等人 [46]对在“实验室环境”而非“真实环境”中评估自动程序修复工具的性能提出了担忧。最后，Durieux 等人 [19] 在一项广泛的评估中发现，许多自动程序修复工具对 Defects4J 基准过拟合。提供完全可执行的训练集和验证集，使生成与验证方法能够遵循更严格的数据分离规范（如 NPR 中的惯例），从而有望减轻基准测试过拟合及其他偏差。

过拟合带来的第二个问题是，当使用较小的评估数据集（包含几十到数百个缺陷）时，少数数据点的性能可能会受到过多关注，从而影响结论的可靠性。值得注意的是，在标准自动程序修复基准上评估神经程序修复系统时，也会受到这一问题的影响。如果数据集能够支持大规模评估，使用更大的评估数据集就能更有效地应对这一问题；这也是接下来几点的重点。

基于测试的评估优于文本评估 如前所述，NPR 在很大程度上遵循神经自然语言处理技术，包括评估实践。然而，与自然语言不同，代码可以执行。在传统的自动程序修复中，执行用于故障定位和基于测试的评估。为了了解测试为何有助于补丁正确性评估，请看这样一种情况：为了修复一个缺陷，表达式 `x` 必须转换为表达式 `x/2`。然而，某个自动程序修复工具将有缺陷的表达式 `x` 转换成了在*逻辑上* 等价、但*词法上不* 等价的表达式 `x * 0.5`。由于两种解法会产生相同的程序输出，测试用例会同等对待它们。然而，精确匹配等词法指标难以捕捉此类语义等价，因此 `x * 0.5` 会被判定为错误的修复。图 2 展示了另一种情况，在这种情况下，词法比较无法正确判定

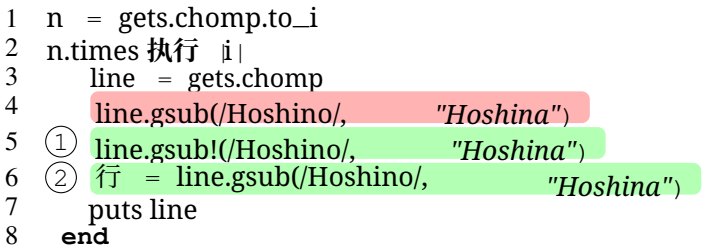


图 2：数据集中 Ruby 缺陷的两种替代补丁。第5行的结果未被使用。补丁 1（第5行）使用会原地修改 line 的方法变体；补丁 2 （第 6 行）将结果重新赋值给 line，从而得到相同结果。这表明，静态判断补丁的正确性（即不通过执行）往往会导致假阴性。

补丁正确性。该缺陷可以通过两种不同方式修复：一种是将 gsub 方法调用的结果重新赋给 line 变量，另一种是使用同一方法的原地变体（由感叹号表示）。

在近期的一项研究中，Haque 等人 [27]比较了用于评估 NPR 模型的静态匹配指标与基于测试（或基于执行）的指标。前者包括精确匹配、BLEU/CodeBLEU [60], 可编译率或 AST 子树重叠度等指标，并在词汇或语法层面进行评估；后者则依赖测试执行（例如，平均通过的测试用例数）。作者发现，基于执行的指标“[. . .] 更能反映程序功能正确性，也比基于匹配的指标更能有效评估模型”。具体而言，作者表明，静态指标（即词汇或语法指标）可能会产生误导。一个简单粗略的复制基线（即将含缺陷代码作为可能的修复方案输出）在 BLEU、CodeBLEU、AST 子树重叠度和编译率指标上的得分，可能高于在修复任务上微调的 CodeT5 [75] 或 PLBART [1]模型，尽管按照设计，该基线从不输出任何正确的修复方案。另一方面，基于执行的指标则清楚地揭示了复制基线的糟糕表现。我们的数据集包含可执行程序和测试用例，因此可以使用基于执行的指标。

提高基于测试的评估效率运行大型测试套件的开销是基于测试的评估的主要缺点之一。例如，Chen、Pei 和 Furia [12] 报告称，补丁验证占其自动程序修复工具运行时间的 92.8%。虽然其他一些自动程序修复工具尝试缩短验证时间（例如，跳过无关的测试用例 [86]，或一次编译多个补丁候选项 [30]），但这需要投入大量工程工作量，而且必须针对每种自动程序修复工具重复进行。此外，目前许多包含测试的数据集都由大型项目中的缺陷构成。要运行这些测试，必须编译整个项目代码库，还可能需要安装大量项目依赖项。除了消耗大量硬盘空间，这也进一步拖慢了测试执行。解决这一问题的一种方法是使用没有或仅依赖少量外部库的小型程序。更快的编译—运行—测试周期每秒可执行多达一百次程序执行（在配备多个 CPU 的机器上），进而使评估数量更多的软件缺陷成为可能。RunBugRun 提供基础设施，可在沙箱中安全地执行和评估数据集中的程序。由于程序相对较小，而且可以并行编译和执行多个程序，其评估速度比 Defects4J 快几个数量级；在 Defects4J 中，庞大的测试套件和构建系统会拖慢编译、执行和测试。

人工评估的规模化问题 通过所有测试用例并不能保证补丁完全正确 [68,58]。遗憾的是，这意味着开发人员需要手动检查其自动程序修复工具生成的每个补丁，并评估其正确性。这是一项繁琐的任务，需要投入大量时间和精力。此外，由于人工评估往往带有主观性，也容易出错 [84, 40], ，通常需要多名标注人员来评估分歧，这进一步增加了工作量。随着测试集和基准测试规模不断扩大，这种人工任务将变得难以为继。

我们认为，拥有强大测试套件的大规模可执行数据集是一种合理的折中方案。覆盖率高的强大测试套件，可以降低错误打补丁后的程序通过全部测试的可能性。虽然这并非完美的解决方案，但人工评估同样可能出错。鉴于 RunBugRun 中包含的程序相对较小，因此可以构建更强大的测试套件，并在合理的时间内运行。我们还可以利用问题通常存在多个解这一事实来增强测试套件：只需付出相对较少的努力，就能通过在随机输入上运行多个解决同一问题的（正确）程序，生成额外的测试用例。如果它们的输出都相同，就可以非常有把握地认为该结果是正确答案，并将其用作测试用例（见第 4 节）。

最后，转向自动化评估不仅能减轻开发人员进行人工评估的负担，还能让自动程序修复系统在数量多得多的软件缺陷上接受评估。这种规模的扩大进而

通过减少过拟合以及过度依赖个别数据点的问题，带来更可靠的性能指标。

真实世界缺陷与人工缺陷 扩展程序修复数据集规模的另一种方法是注入 人工缺陷（例如，通过生成变异体）。生成大量此类人工缺陷相对容易，而收集真实世界缺陷通常需要投入大量精力进行挖掘、筛选和预处理（如下文所述）。不过，真实世界缺陷比人工缺陷更加多样，因为人工缺陷通常来自一组手工设计的变异。事实上，现有研究表明，人工缺陷无法完全替代真实世界缺陷 [61, 3, 28],，这正是 RunBugRun 专注于真实世界缺陷的原因。

尽管如此，RunBugRun 仍可作为这一方向后续研究的基础，例如通过加入设定随机种子的缺陷来扩展训练集，或比较在有无额外缺陷注入的情况下，模型在测试集上的性能。

可执行代码的潜力 最后，但同样重要的是，我们相信，将基于执行的特征整合到 NPR 模型中具有巨大潜力。先前研究已报告了令人鼓舞的结果。例如，Ye 等人的早期研究 [83]表明，通过执行获得的错误消息和堆栈跟踪是宝贵的信息来源，可以提高修复准确率。同样，Drain 等人 [18]报告称，将 Python 堆栈跟踪作为训练数据纳入其 NPR 系统后，性能有所提升。当然，这类信息只能通过代码执行获得。Ye、Martinez 和 Monperrus 的近期研究 [85] 表明，在训练过程中整合测试执行反馈可以提升修复性能。对于少量带有测试的可执行缺陷，他们将基于神经机器翻译的神经程序修复模型与判别器组件结合起来，编译补丁候选项并针对测试套件运行；随后将候选项是否可编译以及通过了多少项测试反馈给整体模型，并将其用作学习信号。借助执行反馈，作者将整体性能提高了 7%。然而，由于可执行缺陷难以获得，这类反馈仅能针对 123 个缺陷收集。我们的数据集提供了数十万个此类可执行缺陷，为这一研究方向开辟了新的机遇。利用执行反馈在程序合成领域也取得了令人鼓舞的结果 [39]。

执行是有用信息的丰富来源。例如，执行轨迹可以揭示程序状态变化。我们期待新一代自动程序修复工具能够利用运行时信息，更好地进行缺陷定位与修复。最后，可执行的真实程序可用于生成任意数量的人工测试用例。

3.2 自动程序修复需要多语言数据集来克服语言偏见

近年来，自然语言处理（NLP）和自然语言理解（NLU）取得了突破，这在很大程度上归功于深度学习。然而，研究人员一再指出，数据集和基准测试（例如 GLUE）中都存在明显的英语偏向 [6]。

自动程序修复领域是否也存在这种偏向？近年发布的 16 个最重要的数据集中，有 11 个包含 Java，5 个包含 Python，3 个包含 JavaScript；而 Ruby、PHP 或 Go 等较不流行但仍十分常见的语言则完全没有出现（表1）。我们希望让更多编程语言进入这一领域，克服当前以 Java 为中心的局面，并推动开发更多支持多种语言、甚至语言无关的自动程序修复系统。这将大幅提高自动程序修复的适用范围，因为大量代码是用 Java、Python 或 C语言以外的语言编写的。随着软件开发所涉及的编程语言日益多样化，软件缺陷可能涉及多种不同的编程语言 [87]。事实上，近期一项研究发现，这种情况出乎意料地常见 [10]。未来的自动程序修复系统若更加重视多语言修复，或许就能更有效地修复这类多语言缺陷。

在自然语言处理领域，不同语言的数据可用性分布极不均衡 [33]。忽视数据可用性较低的语言，尤其是少数语言，可能会威胁它们的生存，并降低语言多样性 [63]。
在代码方面也能观察到类似的模式：与 Ruby 等小众编程语言相比，使用较早出现的语言（如 C语言++）或更流行的语言（如 Python）编写的代码更多。若只将技术进步聚焦于一种或少数几种编程语言，开发人员可能会根据开发工具的质量和可用性来选择编程语言，而非考虑该语言是否适合特定任务。这可能会巩固并强化对既有编程语言的使用，阻碍采用新的或更合适的语言。我们希望，涵盖八种不同编程语言的数据集能够为未来跨越语言边界的自动程序修复研究奠定基础。

3.3 自动程序修复需要经过整理和标注的数据集——执行也能助一臂之力

1 # 项目的一般信息。2 项目 = u'GitPython'34 版权 = u'版权 (C) 2008, 2009 Michael Trier 和贡献者, 2010-2015 SebastianThiel'56 # 所编写文档的项目版本信息，用作以下内容的替代项：7 # /版本 /和 / 发行版 /，也用于

图3：来自 TSSB-3M 数据集的一个修复示例 [62], ，该数据集包含三百万个实例，目前是规模最大的自动程序修复数据集之一。该数据集包含不可执行代码片段。尽管此数据集实例被标记为“可能存在缺陷”，但这一变更（更新版权年份）不太可能修复功能性缺陷。

从提交中挖掘补丁软件缺陷及其对应的人工编写补丁通常从代码仓库中挖掘。不幸的是，仓库提交往往相互交织 [29], ，也就是说，单个提交会处理多个事项，因此可能包含与实际缺陷修复无关的变更。尽管基于提交信息中的关键词匹配这一简单技术在提取修复缺陷的提交时精确率较高，但这类方法的召回率也很低 [4], ，因此难以为当今数据需求旺盛的模型收集大量高质量补丁。挖掘大量提交时，几乎无法避免假阳性。这意味着，大多数基于提交挖掘的自动程序修复数据集都包含一小部分实际上并非软件缺陷或缺陷修复的实例。图3展示了这样一个案例，该图取自 TSSB-3M [62], ，其中更新版权年份被误判为修复缺陷的变更。Lutellier 等人 [47] 通过人工检查其基于提交挖掘的自动程序修复数据集中的一个小样本，发现样本中有 7% 的数据集实例并非实际的缺陷修复。Tufano 等人 [72] 也进行了类似检查，他们在包含 384 个实例的样本中发现，假阳性仅占 2.4%。

拥有一个规模庞大且可执行的数据集有助于解决这个问题：通过执行，我们可以确认这项变化确实修复了缺陷：有缺陷的版本应当存在测试用例失败的情况，而修复后的版本则应当只有通过的测试用例。我们注意到，这仅仅是一个起点，因为还需要采取额外步骤来整理数据。
其中一个步骤是需要对数据进行标注：仅有大量有缺陷/已修复的程序对，其用途有限，因为我们并不知道其中包含哪些类型的软件缺陷。如果这些软件缺陷过于简单或缺乏多样性，那么模型在此类数据集上的性能可能会产生误导。为确保缺陷类型和可见性达到最低限度的多样性，需要以某种方式推断这类信息。此外，了解模型在多种软件缺陷上的细粒度性能，有助于洞察模型表现以及后续如何改进模型（例如，对于涉及运算符的软件缺陷，表现较差的模型，改进方式可能不同于修复函数调用时表现不佳的模型）。数据标注的一些可行方式包括缺陷修复的规模、所采用的修复类型，或者在有执行数据的情况下，有缺陷的程序触发的错误类型。

4 方法论

我们的数据集基于 Project CodeNet [57], ，这是一个大型的多语言数据转储，包含编程竞赛提交。不过，这些数据相当“原始”。因此，我们对其进行了大量预处理和筛选，主要有两个目标：1）确保数据集中的所有程序都能（安全地）执行，并具有足够的测试覆盖率；2）确保数据经过妥善整理和标注，从而在我们关注的一系列指标上构成一个质量和多样性均足够的数据集；3）提供难度适中的基准测试集（既不太简单，也不太困难）以及初始基线。我们按时间顺序介绍所采取的所有步骤，而不是按目标分组，因为这样似乎最为自然。

Project CodeNet 我们的数据集源自 Project CodeNet [57], ，这是一个从两个在线评测网站（AIZU 和 AtCoder）收集的大型编程竞赛提交集合。这些网站会发布编程题目列表，其中包含问题描述以及示例输入和输出。用户可以针对特定题目提交解题程序。网站的评测系统会使用一组不公开的测试用例自动评测提交的程序。Project CodeNet 收录了超过 1,300 万个提交的程序，涵盖 4,000 多道题目。每次提交还会关联其他信息，包括匿名用户 ID、时间戳和评判结果（即该提交是否正确且完整地解决了问题）。如果提交通过了所有测试，就会被接受；否则就会被拒绝。

按语言筛选（整理）首先，我们会移除所有使用非以下编程语言编写的提交：C语言、C++、Java、Python、Ruby、Go 或 PHP。我们选择这些编程语言，是参考了 Code- SearchNet [31], ——一个广受欢迎的代码机器学习数据集，其中包含 Java、Python、Ruby、PHP 和 Go 代码。我们还加入了 C语言和 C++，因为大量提交都是使用这些语言编写的。

筛选可能存在缺陷的程序与已修复程序对（整理）为了从 Project CodeNet 中提取软件缺陷，我们首先按题目和用户 ID 对剩余的提交进行分组。然后，对于同一组中每一个可能的（s_1、 s_2）被拒绝的提交s_1 和被接受的提交 s_2 （即属于同一组的提交，也就是 s_1 和 s_2 属于同一道题目且来自同一用户），我们计算两者之间不同词元的数量。首先将两个提交分别进行词法切分，得到词元序列，再使用基于最长公共子序列的差异算法计算词元级差异。如果差异少于六个词元，我们就将 s_1 其中一个视为有缺陷的提交，将 s_2 另一个视为对应的已修复提交，并将两者加入数据集。此前的研究也采用了类似的方法 [27, 50]。我们将最大词元差异阈值设得相对较低，以便挖掘出的软件缺陷只需少量修改即可修复。这一选择反映出当前最先进的自动程序修复系统仍难以修复需要执行多项修改操作的软件缺陷 [80, 7]。一旦自动程序修复系统能够正确修复例如超过 90% 的软件缺陷，就可以提高这一阈值，从而创建难度更高的数据集。

挖掘测试用例（确保可执行性）所有代码提交都会从标准输入读取一个输入字符串，并将答案输出到标准输出。因此，测试用例是一个字符串对（i， o），其中 i 是输入， o 是预期输出。测试断言，对于给定的程序 p， p（i） $== o$。遗憾的是，Project CodeNet 不包含在线评测网站用于接受或拒绝提交的测试用例。然而，每个问题描述（以 HTML 编写）都包含 1-6 个示例输入/输出对。Project CodeNet 附带了一个脚本，用于从问题描述文件中提取第一个示例对。为了提取所有示例输入/输出对，我们借助自编写的提取脚本，提取出 9,082 个示例对，并将其用作测试用例。遗憾的是，这一数量相当少（平均每个问题大约两个测试用例）。为了进一步增加测试数量，我们添加了 CodeContests[1], 中发现的其他测试用例，该数据集用于训练 AlphaCode [44]。CodeContests 将数据存储为压缩协议缓冲区。解压缩并解码后，由于 CodeContests 和 Project CodeNet 使用相同的问题 ID，因此可以轻松提取这些数据。这又增加了 331,834 个测试用例。最后，对于测试数量较少的问题，我们使用半自动流程生成更多测试用例（见下文）。附录中的图 9 展示了测试用例数量的分布。

筛除存在语法错误的提交（整理）由于我们的目标是进行语义程序修复，因此我们旨在整理出一个由语法有效的语义（或逻辑）缺陷组成的数据集。因此，我们筛除所有包含语法错误的配对（s_1， s_2）（ $s1$ 或 $s2$中任一方存在语法错误均不保留）。对于需要编译步骤的语言（C语言、C++、Java、Go），我们会编译每份提交；若编译失败，则移除对应的配对（s_1， s_2）。对于脚本语言，我们使用相关解释器支持的特殊“仅检查语法”模式来检查语法错误。值得指出的是，也可以采用类似方法（即只保留存在语法错误的提交）来构建语法缺陷数据集。不过，这不属于本研究的范围。

预处理（确保可执行性）大多数提交不需要预处理。然而，我们发现大量 Python 提交是为 Python 2 编写的。我们使用 2to3 工具自动将所有 Python 2 提交转换为兼容 Python 3 的代码（例如，将 raw_input() 替换为 input()）。为确保与较新版本的 Python 兼容，我们考虑了 Python 3.9 版引入的一项变化：最大公约数函数从 fractions 模块移至 math 模块。

执行（确保可执行性）我们会针对相应问题的所有测试用例执行其余提交，并记录执行结果，尤其是程序输出，以及发生错误时的错误消息。所有执行都在沙箱中进行，以保护主机系统免受潜在恶意代码的侵害。我们的沙箱基于 bwrap[2]，Flatpak（一种用于 Linux 的软件包分发和管理系统）也使用该工具。我们对 bwrap 进行配置，使提交无法：a) 连接网络或互联网；b) 读写用户主目录中的文件；c) 读取程序执行所需的预定义系统文件以外的文件（例如编译器和解释器可执行文件、标准库等）。我们使用以下版本的编译器和解释器：a) C语言和 C语言++ 使用 GCC 11；b) Python 使用 CPython 3.9；c) Ruby 使用 CRuby 3.0；d) Java 使用 OpenJDK 17；e) PHP 8.1.2；f) Go 1.18.1；g) NodeJS 12.22。如果提交的执行未能在 3 秒内完成，就会被中止并标记为“超时”。我们还将可用内存限制为 512 兆字节

[1]https://github.com/deepmind/code_竞赛[2]
https://github.com/containers/bubblewrap

通过限制进程的虚拟内存（使用操作系统实现），或设置适当的虚拟机/解释器参数（Java虚拟机、NodeJS），来施加内存限制。我们总共记录了超过 1.3 亿次执行；这耗时数周。

执行后的提交与测试用例筛选（整理）执行结果为进一步筛选提供依据。如果出现以下情况，我们会移除任意一对（s_1, s_2）：a） s_1 通过了所有测试用例，或在任何测试用例上超时；或 b） s_2 未通过任何测试用例，或在任何测试用例上超时。此外，我们会筛除以下测试用例：a）总体通过率低于 45%；或 b）超时率高于 12%（即超过 12% 的提交在接收该测试用例的输入时超时）。在选择这些阈值时，我们力求在尽可能保留更多测试与筛除通过率低或超时率高的测试之间取得良好平衡。我们发现，45% 和 12% 能够实现较好的权衡。通过率和超时率的分布情况（附录中的图8）或许能进一步说明为何选择这些阈值。

通过执行生成测试用例（确保可执行性）即使添加了描述文件和 AlphaCode 的 CodeContests 中的所有测试用例，仍有一些问题没有测试用例或测试用例很少。对于这些问题，我们以半自动方式生成额外的测试用例。针对每个问题，我们编写脚本生成随机输入（遵循问题规范）。随后，我们抽取 10–15 个（视问题而定）随机的正确提交（根据在线评测系统的判定），并将生成的输入提供给这些提交。如果所有抽取的正确提交都产生相同的输出，我们就认为该输出正确，并将其与生成的输入一并添加到测试用例列表中。通过这种方式，我们为 111 个问题生成了 600 多个额外的测试用例。

标注（整理）我们会自动为筛选后的错误，即错误/修复提交对（s_1, s_2），标注一个或多个标签。我们的标注系统基于规则。我们使用 TreeSitter 解析库（ s_1 和 s_2 均由该库解析；该库支持我们数据集中的所有语言）进行解析。随后，我们找出解析树中受s_1 与 s_2 之间变化影响的所有节点（即某个特定节点被添加、删除或替换）。具体做法是先对 s_1 和 s_2进行词法切分，然后计算基于最长公共子序列的词元级差异。对于每个变更、插入或删除的词元，我们会查找解析树中与之对应的节点。接下来，将变更后的节点与一组手动编写的规则进行匹配。若某条规则匹配成功，就会为该软件缺陷标注相应的标签。例如，假设为了修复 s_1中的软件缺陷，在 s_2 中的循环内添加了一条 break 语句。变更节点集合将是一个单元素集合，其中仅包含一个类型为 statement_break 的节点。此时找到一条匹配规则：由于 a) 添加了一个节点，且 b) 该节点的类型为 statement_break，因此规则匹配成功。于是，该软件缺陷被标注为 control_flow.break.add。标签采用层级结构，即按共同前缀分组。例如，所有与控制流相关的标签（如跳出循环、跳过一次迭代、添加或删除 if-else 分支）都具有共同前缀（标签层级的示意图见附录中的图14）。由于不同语言的解析树可能不同，同一规则通常需要针对不同语言分别实现。我们的“变更分类体系”包含 136 个不同的叶级标签和 16 个词干标签（即前缀）。在某些情况下，变更属于“单一类型”，无法与任何规则匹配。约 9% 的提交属于这种情况。平均而言，每个软件缺陷被标注 2.3 个标签；25% 的数据实例有一个标签，26.8% 有两个标签。

排除不稳定的提交（整理）先前的研究发现，不稳定测试会对缺陷定位产生负面影响，并降低修复成功率 [59]。我们检查测试集和验证集中的所有提交是否存在不稳定性，即重复执行时输出是否会随机变化。为此，我们多次（至少五次）运行属于这些集合的所有已修复的提交。如果任何一次运行未能通过所有测试用例，就将其视为不稳定，并从数据集中移除。我们发现，不稳定性的主要来源是 C语言 和 C++ 中的内存错误（例如双重释放或无效内存访问）。有一个 Python 提交中的不稳定性源于对伪随机数的不当使用（未设定随机种子）。

数据拆分（基准测试）为了确保比较结果具有可比性，我们将数据拆分为预先定义的训练集、测试集和验证集，数据集的未来用户应当使用这些集合。大多数提交使用 C++ 和 Python 编写。不过，为了平衡这些编程语言占比过高的问题，我们确保测试集中的语言分布更加均衡。在测试集中，占比最高与最低的编程语言（C++ 和 Python 与 Go、JavaScript 和 PHP）之间的比例约为 10 倍，而在整体数据中这一比例超过 100 倍。如前所述，每个提交都关联一个匿名用户 ID。我们以这样的方式拆分数据：属于同一问题和同一用户的多个数据集实例绝不会被拆分到不同集合中。

我们决定按问题维度进行拆分。也就是说，解决同一问题的提交会被分配到三个集合中。由于这些提交解决的是同一个问题，它们可能具有一定的相似性（但低于去重阈值）。我们选择这种设置，是因为自动程序修复的核心思想之一是将正确代码“移植”到含缺陷代码中。从某种意义上说，我们测试的是自动程序修复系统从训练集中学习和掌握修复模式的能力

[3]https ://tree-sitter.github.io/tree-sitte r/

并将这些模式应用于测试集。这在某种程度上类似于整形手术假说 [5]，即可以从训练集中的程序中学习修复成分。

我们考虑过另一种方案，即仅在测试集中纳入未见过的问题，以进一步强调泛化能力。不过，我们认为，以当前自动程序修复（APR）/神经程序修复（NPR）的发展水平，这种设置挑战过大。我们计划未来采用这种格式构建一个后续数据集，在 RunBugRun 的性能开始趋于饱和时推出。

去重（整理） 虽然训练集和测试集中有针对相同问题的提交，我们仍希望避免数据泄漏。为此，我们采取以下去重措施：a）从整个数据集中删除所有完全重复的内容（忽略空白字符）；b）删除训练集与测试集之间，以及训练集与验证集之间的跨集合近似重复项。我们使用三元组相似度，在八种不同编程语言的数十万条提交中高效查找近似重复项。

基线（基准测试） 为了让未来的数据集用户能够以预期性能为参照，我们提供两个简单的基线：一个基于 CodeT5 [75] 的神经程序修复（NPR）系统，以及一个 G&V 基线。我们选择 Cardumen [52] 作为 G&V 基线，因为它是 QuixBugs [82], 上最成功的生成与验证系统之一。QuixBugs 是一个在代码类型和缺陷方面（即算法实现中的简单缺陷）与我们的数据集相似的基准测试集。有关评估方法的详情，请参阅第 5.2.1 节。

5 结果与讨论

我们的数据集包含超过 450,000 个软件缺陷，即从编程竞赛网站的提交中提取的有缺陷/已修复的程序对。每个缺陷都关联着：i) 一个唯一的缺陷标识符；ii) 一个语言标识符（例如，Java）；iii) 一个或多个可选标签（9% 的缺陷不属于任何现有标签）；iv) 一组以输入/输出对形式表示的测试用例；v) 有缺陷的程序引发的零条或多条错误消息（见图 1、12 和 13）。

我们探讨以下主要研究问题：

RQ1. 数据集统计： 我们希望了解该数据集最重要的关键数据。直言不讳地说，我们想知道“其中包含什么”。具体而言，我们尝试回答以下问题：

RQ1.1. 编程语言的分布情况如何？RQ1.2. 变更数量的分布情况如何？RQ1.3. 最常见的错误标签有哪些？RQ1.4. 测试执行期间最常出现哪些错误和异常？

RQ2. 基线： 第二个研究问题关注基线性能。我们的目标是为数据集的未来用户提供衡量其工具性能的标尺。最后，对于 NPR 基线，我们希望了解能否在一种语言中学习修复方法并将其应用于另一种语言（即进行迁移）。鉴于编程语言分布不均，这一点尤为重要：某些语言（如 C++ 或 Python）的数据量很大，而其他语言（如 PHP 或 Go）的数据量则很少。具体而言，我们提出以下问题：

RQ2.1. 所选基线在该数据集上的表现如何？性能如何随编程语言以及变更数量/代码块而变化？
RQ2.2. 性能如何随标签而变化？NPR 和 G&V 基线各自有哪些优势与不足？RQ2.3. 软件缺陷能否从一种语言迁移到另一种语言？也就是说，能否在一种语言中学习软件缺陷，并在另一种语言中修复它们？
研究问题2.4. 生成的候选项数量会如何影响性能？

5.1 数据集统计（RQ1）

5.1.1 语言分布（RQ1.1）

我们的数据集包含八种不同的编程语言（C++、C、Python、Java、Ruby、PHP、JavaScript、Go）。遗憾的是，使用 C/C++ 和 Python 编写的提交数量明显偏多，而 PHP、JavaScript 和 Go 的提交数量则很少。为缓解这种不平衡，测试集中代表性不足的编程语言所占比例相对更高（至少有 1,000 个实例）。正如我们将在后文展示的，现代 NPR 模型能够学习一种语言中的修复模式，并将其应用于其他语言（RQ2）。因此，尽管存在这种不平衡，我们认为

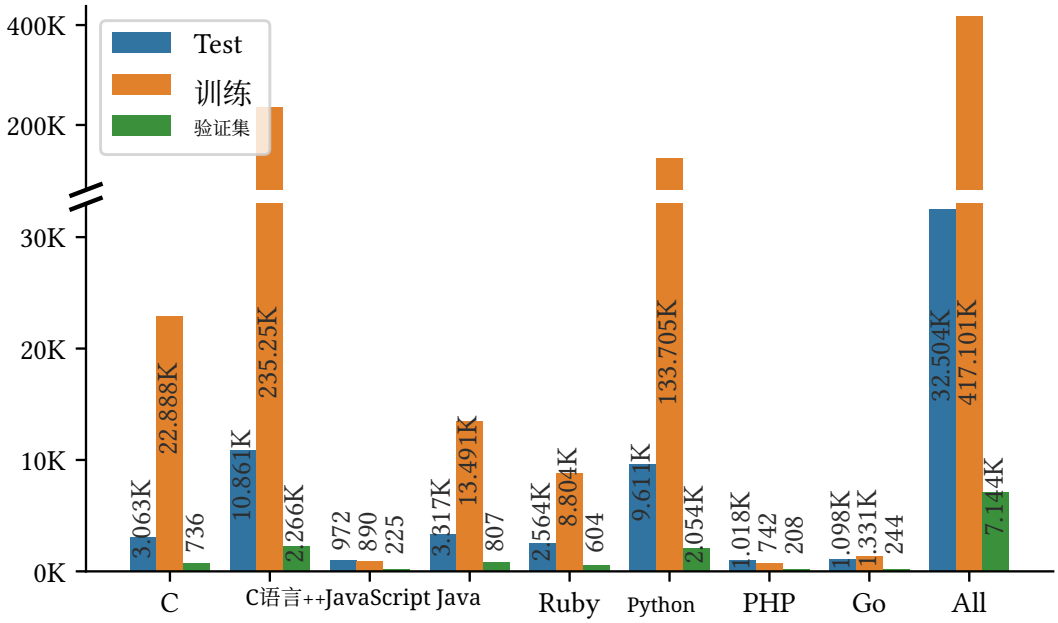


图 4：训练集、测试集和验证集的语言分布。

我们的数据集仍为未来的多语言APR系统作出了重要贡献。图 4 展示了训练集、测试集和验证集中的语言分布。C++ 和 Python 是最常见的语言，训练集中各有超过 100,000 个实例，其次是 C，超过 23,000 个。从规模来看，Java 和 Ruby 处于中等范围，训练集中的实例数分别超过 8,000 和 13,000。PHP 和 JavaScript 的训练集均略少于 1,000 个实例，Go 则略多于 1,000 个。

> **RQ1.1 的回答**
> 语言分布严重偏向 C/C++和 Python。JavaScript、Go 和 PHP 是出现频率最低的编程语言。我们通过提高测试集中低频编程语言所占的比例来弥补这种不平衡，使出现频率最高和最低的编程语言之间的比例约为 10 倍（在整体数据中，这一比例约为 125 倍）。

5.1.2 变更数量（RQ1.2）

如前所述，数据集实例的缺陷版本与修复后的版本之间发生变化的语言令牌数量为 1 至 6 个。具有特定变更数量的实例数会随着变更数量的增加而减少。近三分之一的案例中，差异仅为一个词元；而变化涉及六个词元的案例仅占数据的 8.6%。除 JavaScript 和 Python 的中位数为 3 外，所有编程语言的变更数量中位数均为 2。图5展示了数据中变化词元数量的分布。行代码块的分布也类似，即发生变化的连续行块数量。约 75% 的软件缺陷属于单代码块缺陷。20% 的变更包含两个代码块，只有 5% 包含两个以上的代码块（即 3 至 6 个代码块）。

> **RQ1.2 的回答**
> 数据集中的软件缺陷中，近 30% 只需变更一个词元即可修复。变更数量越多，具有该变更数量的数据所占比例越低。仅有 25% 的软件缺陷需要多代码块变更。

5.1.3 缺陷标签（RQ1.3）

如前所述，我们使用基于规则的方法自动为所有软件缺陷添加标签。一个缺陷可以有多个标签，且标签具有层级结构。例如，如果修复某个缺陷需要修改 if 条件，则使用标签 control_-

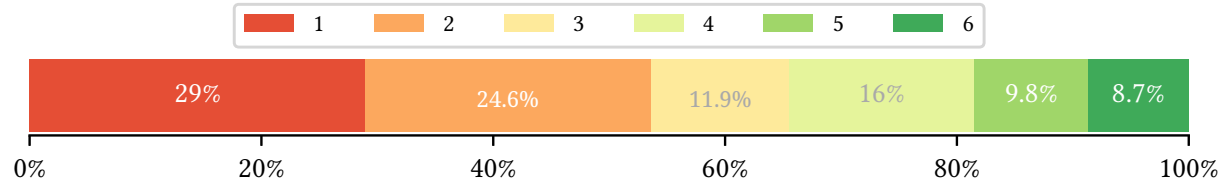


图5：变更数量的分布 从单项变更（最左侧）到六项变更（最右侧）).

flow.branch.if.condition.change。具有相同前缀的标签表示相近的含义。删除 if-else 语句中的 else 分支会标记为 control_flow.branch.else.remove，因为这可能会改变控制流和分支。类似地，添加 return 语句也会改变控制流。不过，这与分支无关，因此使用的标签是 control_flow.return.add。标签以一个或多个动作结尾，用于指明变更类型。动作可以是 .change、.add 或 .remove。附录以图示形式列出了所有标签（图14）。

最常见的标签 表2按编程语言概述了最常见的标签。不同编程语言之间的差异相对较小。在八种编程语言中的六种里，需要更改函数或方法调用参数的缺陷最为常见（call.arguments.change）。另外两种编程语言（C++和 Go）中，最常见的是对 if 条件的更改。总体而言，这类缺陷十分常见，在所有编程语言中都属于最常见的四种变更类型之一。这些结果与以往研究一致。具体而言，Pan、Kim 和 Whitehead [56], 分析了七个大型 Java 开源项目中的缺陷修复模式后发现，与方法调用和 if 条件相关的修复模式最为常见（“使用不同的实际参数值调用方法”和“更改 if 条件”）。

标签 io.output.change 是除 Ruby 之外所有编程语言中出现频率最高的六个标签之一。当程序输出发生变化时（例如，更改打印格式字符串，或添加、删除 print()/printf()/puts() 调用），就会分配此标签。字面量（整数或字符串）的变更也很常见，在五种编程语言的前六个标签中都有出现。最后，我们发现，最常被修改的运算符是比较运算符（expression.operator.compare.change）。所有标签的概览请参见附录中的图14。

> **RQ1.3 的回答**
> 方法和函数调用的参数以及 if 条件的变更，是所有编程语言中占主导地位的缺陷类型。这与以往的研究结果一致。此外，字面量变更以及改变输出的变更（例如，print 调用或格式字符串变更）也很常见。最常变更的运算符是比较运算符（例如，<、>=、!=）。

5.1.4 错误与异常（RQ1.4）

什么是错误? 如前所述，我们会对所有测试输入运行数据集中的所有程序。程序可能会：a）输出正确答案；b）输出错误答案；c）超时；或 d）产生运行时错误。程序执行期间是否发生错误，并不总是显而易见。我们采用一种简单的启发式方法：如果 a）执行因操作系统级信号（例如 SIGSEV 或 SIGABORT）而终止，或 b）程序向标准错误（即 stderr）输出内容，并且通过返回非零退出状态表明执行失败，则判定此次执行产生了错误。程序可能产生错误，但仍向标准输出输出正确答案。在这种情况下，正确输出会被忽略。对于能够始终通过异常处理错误的编程语言，我们可以提取异常名称并准确统计其发生频率。这些语言包括 Python、Ruby 和 Java。对于 C++ 和 JavaScript，错误不一定总是包含异常类型。我们报告错误消息中发现的所有异常。PHP 解释器会将警告和错误输出到标准输出，而不是标准错误。这使得错误的判定变得困难且不可靠。因此，我们对 PHP 错误的分析仅限于将程序输出与常见错误消息模式进行匹配。对于 C/C++和 Go，错误的判定更加可靠。不过，由于这些语言没有异常系统（或者在 C++的情况下，程序通常会在未抛出异常时失败），我们同样采用将错误消息与一组常见消息模式进行匹配的方法。图6展示了数据集中一个会引发类型错误的 Python 缺陷。表3列出了执行期间最常见的异常及其相对发生频率。

表2：最常见的　　各语言缺陷标签。频率是与特定　　c
标签。由于多个实验室签，因此频率之和不必达到 100%。

语言	标签	频率
C	调用参数变更	41.66%
	字符串字面量变化	35.26%
	输入输出输出变更	32.98%
	控制_flow.branch.if.condition.change	19.62%
	表达式二元运算变更	15.06%
	数字字面量变更	14.92%
C++	控制_flow.branch.if.condition.change	21.41%
	输入输出输出变更	14.96%
	数字字面量变更	14.96%
	表达式二元运算变更	14.56%
	expression.operator.compare.change	12.92%
	控制_流循环for条件变化	12.09%
JavaScript	调用参数变更	37.90%
	输入输出输出变更	21.13%
	表达式二元运算变更	19.74%
	控制_flow.branch.if.condition.change	17.35%
	字符串字面量变化	15.62%
	标识符变更	14.85%
Java	调用参数变更	23.81%
	控制_flow.branch.if.condition.change	21.02%
	表达式二元运算变更	19.51%
	输入输出输出变更	17.96%
	标识符变更	16.64%
	expression.operator.compare.change	15.24%
Ruby	调用参数变更	25.60%
	控制_flow.branch.if.condition.change	19.45%
	赋值语句值变化	16.54%
	调用.add	16.50%
	标识符变更	15.85%
	字符串字面量变化	11.52%
Python	调用参数变更	37.51%
	控制_flow.branch.if.condition.change	19.60%
	标识符变更	16.31%
	输入输出输出变更	15.49%
	赋值语句值变化	15.34%
	表达式二元运算变更	13.59%
PHP	调用参数变更	19.82%
	控制_flow.branch.if.condition.change	19.72%
	输入输出输出变更	18.65%
	赋值语句值变化	17.43%
	字符串字面量变化	15.55%
	表达式二元运算变更	11.79%
Go	控制_flow.branch.if.condition.change	24.47%
	调用参数变更	24.13%
	expression.operator.compare.change	19.19%
	表达式二元运算变更	19.12%
	输入输出输出变更	16.87%
	控制_流.循环.for.条件.变化	16.69%

```
1 def is_triangle(x, y, z): 2 x, y, z = sorted([x, y, z]) 3 4 return x**2 + y**2==z**25 6num = int(input()) 7 for iin range(num): 8 x, y, z = map(int, input().split()) 9 triangle =is_triangle(x, y, z) 10 iftriangle: 11 print('是') 12 else: 13 print('否')
```

```
回溯(most recent call last):文件"file.py",第8, in <module>triangle = is_triangle(x, y, z)文件"file.py",第2, in is_triangle x, y, z = sorted(x, y, z)类型错误: sorted 预期接收 1 个参数，但实际收到 3 个
```

图6：sorted 函数接受一个列表类型的参数（缺陷 #885329）。由于传入了多个参数，因此引发类型错误。堆栈跟踪（右侧）立即显示了缺陷所在位置。

动态语言对于 Python、Ruby 和 JavaScript，这些语言中最常见的异常类型与其动态特性有关。此类错误包括：a）引用未定义的变量、方法、函数或属性（名称错误、引用错误、无方法错误、属性错误）；b）类型错误（类型错误）。与 Java 类似，在 Python 中，由数组越界访问或使用无效键进行映射/字典查找导致的索引错误和键错误相对常见（约占 17%）；而在 Ruby 和 JavaScript 中，这类错误较少发生（不到 1%）。造成这种差异的可能原因是：Python 会在越界访问和查找时引发异常，而 Ruby 和 JavaScript 则返回空值（Ruby 中为 nil，JavaScript 中为 undefined）。对于 PHP，我们发现了类似的模式：最常见的错误消息是“使用了未定义的常量”，例如由引用未定义变量所导致。参数错误也是 PHP 程序中另一类常见错误（例如，函数调用时传入了错误数量或错误类型的参数所致）。遗憾的是，由于上述限制，我们无法可靠地量化 PHP 中各类错误的频率。

静态语言 C/C++、Go 和 Java 都是静态语言。名称错误会在编译期间被捕获，因此不会在运行时发生。对于 Java、C++ 和 Go，我们发现与越界访问相关的错误是最主要的类型。在 Java 中，排名前三的异常类型都与无效边界有关，占所有抛出异常的 50% 以上。同样，对于 C++，std::out_of_range 是最常被抛出的异常。由于 C/C++ 中的数组访问通常不会进行检查（此时程序会崩溃而不会抛出异常），与边界相关的软件缺陷数量实际上可能更高。最后，Go 的错误消息也呈现出类似的情况。其中，71% 的错误消息符合“索引超出范围”模式，6% 符合“切片边界超出范围”模式。

Java 程序执行期间发生空指针异常的频率仅为 2.59%，低得令人惊讶；先前的研究发现，Android 应用程序中多达 27.7% 的崩溃与此类错误有关 [15]。

内存错误 C语言和 C语言++ 属于不安全的编程语言，需要手动管理内存。正如预期，在执行用这些语言编写的程序时，我们看到了大量与内存相关的问题。对于 C语言++，在超过 30% 的案例中，我们发现了 malloc、free()、双重释放、无效指针或栈破坏等关键词。C语言中的这一比例更高，超过 90% 的错误消息都包含上述关键词之一。不过，由于 C语言没有异常系统，而且内存问题是少数会报告到标准错误中的错误之一，因此这些数字需要谨慎看待。由于 C语言中的内存访问通常不会经过检查（C语言++中也并非总是如此），内存错误可能由多种原因引起，其中也包括无效边界。

RQ1.4 的回答
错误和异常的类型与语言采用静态类型还是动态类型密切相关。对于动态类型语言（Python、JavaScript、Ruby、PHP），变量名或方法名解析错误最为常见。对于静态语言（C语言/C语言++、Java、Go），大多数错误都与无效边界有关。C语言++中内存错误很常见，在 C语言中尤为如此。就动态语言而言，我们注意到，语法程序修复与语义程序修复之间的界限相当模糊，因为许多静态语言中的错误（即编译器错误）在此处由语法修复处理时，实际上会在运行时发生（例如，使用未定义的变量、方法或函数）。

表3： 执行期间最常出现的异常。频次按程序/缺陷统计。

语言	异常 Exception/Error	Freq.
Python	名称错误	33.28%
	类型错误	25.77%
	索引错误	16.64%
	值错误	9.68%
	属性错误	5.57%
	除零错误	2.69%
	文件结束错误	2.22%
	语法错误	1.21%
	键错误	0.79%
	模块未找到错误	0.79%
Java	数组索引越界异常	36.99%
	输入不匹配异常[2]	15.46%
	字符串索引越界异常	9.59%
	无此元素异常[2]	8.81%
	算术异常[1]	5.23%
	数字格式异常[1]	5.09%
	类未找到异常[1]	3.03%
	索引越界异常	2.89%
	空指针异常[1]	2.59%
	负数组大小异常	2.05%
Ruby	无方法错误	48.11%
	名称错误	27.61%
	类型错误	11.44%
	参数错误	6.28%
	除零错误	2.54%
	加载错误	1.07%
	索引错误	0.68%
	系统栈错误	0.40%
	浮点域错误	0.32%
	内存不足错误	0.24%
JavaScript	引用错误	51.94%
	类型错误	39.72%
	范围错误	1.39%
	语法错误	0.83%
C++	std::out_of_range	32.62%
	std::length_错误	11.39%
	std::bad_分配	10.35%
	std::invalid_参数	1.07%
	std::logic_错误	0.42%
	std::bad_array_new_length	0.32%
	std::运行时_错误	0.03%
	std::bad_函数_call	0.03%

[1] 在 java.lang 中;

[2] 在 java.util 中

5.2 基线（RQ2）

如前所述，我们使用 NPR 基线和 G&V 基线对该数据集进行评估。神经网络基线基于 CodeT5 [75]。对于 G&V 基线，我们采用 Cardumen [52]。此次评估旨在为该数据集未来的用户提供一个参考点，以了解他们可以预期其工具达到的性能。

5.2.1 评估

对于 NPR 基线，我们使用 HuggingFace 的 transformers 库 [77] 对公开可用的 codet5-small 模型检查点在训练集上进行微调。我们将修复过程建模为一个序列到序列翻译问题（从有缺陷的代码到修复后的代码），也就是说，模型学习同时定位和修复 缺陷。因此，我们既不使用错误位置标记（例如 SequenceR [13]），也不假设能够完美定位。我们在单个 NVIDIA RTX 3090（显存为 24GB）上训练五个周期，学习率为 1e-4，梯度累积步数为 2，批量大小为 16。对于推理（即测试集评估），除非另有说明，否则我们为每个实例生成五个输出（即五个候选项），并采用具有六个束的束搜索。先前的研究表明，对于当前任务，应优先使用束搜索而非贪心解码 [27]。为估计候选项数量对性能的影响，在一项实验中，我们为每个缺陷最多生成十个候选项（图7）。在训练和推理期间，我们在所有提交内容前加上一个特殊的语言标识令牌（例如 <python>）。这一思想取自 PLBART [1], ，后者是一种成功的多语言代码模型。

Cardumen 只能处理 Java 代码中的缺陷，并且需要 JUnit 测试用例。我们会自动将所有测试用例转换为 JUnit4 测试，并且只在测试集中的 Java 缺陷上进行测试。为了能够在数千个程序上运行 Cardumen，我们不得不将最长运行时间缩短至 10 分钟，并将生成次数限制为 100（- maxtime 10 -maxgen 100）。对于生成与验证工具而言，这样的时间预算异常短，因为其时间预算通常以小时计 [52]。此前的研究已指出，生成与验证系统的运行时间很长 [54]。这不仅限制了生成与验证方法的实用性，也使得在大量软件缺陷上评估这些工具变得非常困难。

在故障定位方面，我们使用 ASTOR 的[4] 新型 Flacoco [67] 故障定位引擎，而不是 GZoltar [11], ，因为后者与近期的 Java 版本不兼容。

5.2.2 基线性能（RQ2.1）

G&V 基线 Cardumen 成功（即合理地）修复了约 3,300 个 Java 软件缺陷中的 157 个。这一结果相当令人失望。然而，如前所述，为了处理如此大量的软件缺陷，我们不得不将时间预算缩短到异常低的水平。还需注意，生成与验证系统基于冗余假设运行，即假设修复所需的正确成分可以在待修复程序中找到。由于这里处理的是非常小的程序，因此冗余很少，甚至不存在。先前的研究已讨论过这一问题 [81]。

NPR 基线 基于 CodeT5 的 NPR 模型表现出人意料地好，在每个软件缺陷有五个候选项的情况下，成功修复了 59.8% 的提交（涵盖所有编程语言）。

性能与变更数量 我们计算了变更数量（修改的词元数）与基线性能（合理率）之间的皮尔逊相关系数。正如预期，两个基线都呈现显著的负相关，即变更数量越多，性能越低。根据 $r = -0.996$，这一相关性在神经网络基线中强于 G&V 基线，其中 $r = -0.6823$。神经网络基线的性能从单次变更修复的 68.7% 降至需要修改六个词元的修复的 42.8%（每个缺陷有五个候选项）；对于 G&V 基线，相应数值分别为 2% 和 0.09%（仅针对 Java 软件缺陷）。

不同编程语言下的性能 Cardumen 是我们的 G&V 基线工具，仅支持 Java。因此，我们重点关注 NPR 基线。首先，借助 $r = 0.33$，我们发现，对于特定语言，训练样本数量与性能之间存在中等程度的相关性。每个缺陷有五个候选项时，Python 的性能（即合理性）最高（69%），其次是 C语言（64%）和 Java（62%）；难度最高的语言是 Ruby（47.6%）和 Go（47.8%）。更多详情请参阅表 5。

[4]ASTOR [51] 是围绕 Cardum 构建的框架 en

RQ2.1 的回答
Cardumen（G&V 基线）的可能修复的软件缺陷仅占 5%，表现相当不佳。相比之下，基于 CodeT5 的 NPR 模型表现出乎意料地好，可能修复了近 60% 的软件缺陷。正如预期，软件缺陷越复杂（令牌变更数越高），性能越低。C语言、Python 和 Java 的性能最高（> 60%），而 Ruby 和 Go 则排名靠后（< 50%）。

5.2.3 不同标签下的性能——优势与劣势（RQ2.2）

错误标签使我们能够进行逐标签分析。我们考虑两种分析。首先，计算每个标签的合理性比率，以显示特定标签的缺陷被修复的比例。其次，分析优势与劣势。在此，我们计算特定标签的合理性相对于 其在数据集中的出现频率。我们将其称为标签的 r值。下文相应章节将更详细地说明如何获得这些 r 值。

不同标签下的性能 我们首先考察 CodeT5 在测试集中支持数达到 30 或以上的各个标签上的整体性能（具体而言，即可能修复的错误中带有给定标签的百分比），并将其汇总为若干类别。对于细粒度标签，性能从 90% 或以上（添加一元引用、添加 PHP 标签）到低于 20%–25%（添加 continue 关键字或更改某些运算符）不等，表明不同类型的软件缺陷之间存在较大差异。将标签汇总为顶层类别（表 4）后，我们发现，与变量相关的类别（标识符，52%；赋值，48%；变量访问，40%）表现似乎较差（变量声明除外，为 66%）。函数返回值和预处理器类别的表现也较差（34%）。控制_流类别（58%）略低于所有软件缺陷的平均水平（60%）；另一方面，函数调用（63%）、字面量（66%）、输入/输出（76%）和类型转换（81%）的表现则高于平均水平。值得注意的是，尚未标注的缺陷表现略低于平均水平（53%），这表明未标注缺陷的“长尾”整体上更具挑战性。

请注意，性能与数据量的相关性并不高（假设训练集上的分布相同）。相关系数仅为 0.09，这表明还有其他因素在起作用，例如各个标签本身的难度（例如，php_tag 是性能最高的标签，但相对少见）。

对于 Cardumen，我们发现许多标签下完全没有合理修复的缺陷。这些标签包括添加函数或方法调用、为字符串补上缺失的换行符，或添加类型转换。效果最好的标签与运算符有关。移除运算符的合理性比例最高，接近 12%；更改算术运算符的比例为 9%。
接下来，我们将分析形式化，以检测弱项和强项，同时考虑数据规模。

表 4：按标签划分的性能 （合理性比例），按宏平均和微平均分组汇总我们的 NPR 基线 o
平均值，分别。

	Macro	Micro
类型转换	78.6%	81.3%
IO	75.5%	75.5%
字面量	51.8%	66.1%
变量声明	57.0%	65.5%
Call	64.2%	63.3%
All	59.8%	59.8%
Misc	70.0%	59.4%
表达式	55.1%	59.3%
控制流	50.8%	57.8%
无标签	52.5%	52.5%
标识符	49.8%	51.9%
赋值	48.0%	47.9%
变量访问	39.6%	39.6%
函数	34.1%	34.1%
预处理器	33.7%	33.7%

优势与劣势 我们采用以下方法来检测此类弱项和强项。设 $f(L, l)$ 为标签 l 相对于多重集 L 中标签的相对频率。那么，对于每个已修复缺陷的标签 l （来自

测试集），我们计算其商

$$r_l = \frac{f(L_{plausible}, l)}{f(L_{test}, l)}$$

其中，L_{test} 是测试集中所有标签构成的多重集，$L_{plausible}$ 是可能修复的错误集合中所有标签构成的多重集。我们根据 r_l对 $L_{plausible}$ 中的所有标签进行排序。我们将排名前k 的标签（在分析中使用 $k = 5$ ）视为“强项”，将排名后 k 的标签视为“弱项”。一般来说，$r \gg 1$ 表示强项，而 $r \ll 1$ 表示弱项。为获得较为可靠的数据，我们只针对测试集中绝对频次不低于 30 的标签计算并分析 r_l 。

强项 对于我们的 NPR 基线，得分最高的标签是 r-score为 `expression.operation.unary.reference.add`。如果修复缺陷时涉及在 C 语言中添加引用运算符，就会分配此标签，C++、Go，例如，`scanf("%d", &mnt[i]);`。其他强项包括：

- 更改字符串标签的大小写（例如 `System.out.println(m + "X"'` [redacted] `: + "=" + x *m),`
- 添加缺失的 PHP 标签（[redacted]，
- 将赋值运算符替换为比较运算符（例如，if (a`[1]` [redacted] 3)),
- 添加类型转换（例如，`cout << (int)` [redacted] `<< endl;`),
- 更改变量声明类型（例如，`intlong l` [redacted] 最终
- 将 if 分支更改为 else-if 分支（`elif y` [redacted] `base:`）。

对于 Cardumen，涉及运算符反转的缺陷修复最为成功，例如，将 `<` 改为 `>`、`!=` 改为`==` 等；其次是要删除、替换或更改简单二元运算的缺陷（例如，长余数 `= x % 11 * 2;`、`System.out.println(n * [redacted]);` 以及`System.out.println(10 *b *+a + c);`）。

弱项 NPR 基线最大的弱项是添加 if 分支（例如，`if j <m:`），以及在循环中添加 [redacted]`tinue`。尽管更改字符串 [redacted] 标签位列得分最高的 15 个标签之中，但该模型似乎难以处理整数字面量，这可能是因为这类变更通常非常具体，缺乏通用规则（例如，`#define N 20000002`）。整数字面量的变更通常发生在数组维度声明或赋值中（例如，`int P[` [redacted] `];`）。这两类变更的标签，即添加、更改或删除赋值，或更改数组维度，其 r 值同样较低。

由于 Cardumen 的性能相对较差，许多标签对应的缺陷完全没有修复方案。例如，我们发现 Cardumen 未能修复任何需要添加方法或函数调用的缺陷。在可能修复的缺陷中，也没有涉及添加类型转换的缺陷。此外，它在修复有问题的 for 循环初始化表达式或条件时表现也不佳。

> **RQ2.2 的回答**
> CodeT5 模型擅长修复特定的运算符错误（缺少引用运算符，或使用 = 而非 ==），以及数值类型转换和错误（数值）类型的声明。该模型似乎难以处理某些需要插入 continue 关键字或新增 if 分支的控制流错误。Cardumen 可能修复的缺陷大多涉及某种简单的表达式修改，例如反转比较运算符或替换算术运算符。对于这类简单缺陷之外的许多其他类型，Cardumen 则难以修复；例如，它无法修复任何需要添加函数调用或类型转换的缺陷。

5.2.4 跨语言知识迁移（RQ2.3）

一般来说，当模型能够将从一项任务中学到的知识应用于另一项任务时，就会发生知识迁移。当某项任务的数据稀缺，而另一项类似任务的数据并不稀缺时，这种策略尤其有用。例如，为了提升数据可用性较低的编程语言的机器翻译性能，Zoph 等人 [89] 首先在一种相关的“高资源语言”（即数据可用性高的语言）上训练模型。近期研究表明，对于多种软件工程任务（例如代码摘要或函数命名），类似的方法也可能适用于低资源编程语言 [2]。在这里，我们关注的是，缺陷及其修复能否从一种语言迁移到另一种语言。具体而言，我们想了解，在数据较少的语言（例如 PHP 或 Go）中修复缺陷的模型，能否通过在数据丰富的其他语言（例如 C语言++或 Python）上训练而受益。

表 5：采用不同训练方案的 NPR（CodeT5）基线结果（合理率，百分比）：训练数据包括所有编程语言（all）、除测试语言外的所有编程语言（no），以及仅目标语言（only）。所有数值均四舍五入至小数点后一位，并使用每个缺陷 5 个候选项计算得出。

语言	All	No	Only	迁移
C	**64.4**	60.6	50.0	+14.4
C++	**54.9**	26.3	53.7	+1.2
Java	**62.9**	50.8	40.9	+22.0
Python	**69.0**	26.2	66.8	+2.2
Ruby	**47.6**	13.7	34.0	+13.6
Go	**47.8**	1.1	16.0	+31.8
PHP	**55.7**	24.2	21.2	+34.5
JavaScript	**44.5**	34.4	16.2	+28.3

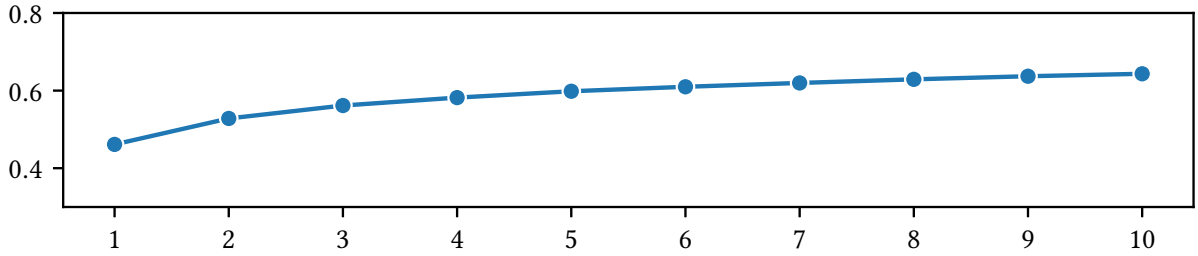


图7：在所有编程语言上进行训练和评估时，基于 CodeT5 的模型在不同每个缺陷的候选项数（x）下的合理率（y）。

为了评估语言之间的这种知识迁移效应，我们采用不同的训练机制训练了 17 个 NPR 模型。其中一个模型使用整个训练集（即所有语言）进行训练。然后，对于八种语言中的每一种 l，我们再训练两个模型：一个模型不使用该语言的训练实例 l，另一个模型仅 使用这些语言的实例 l。结果见表 5。语言 列指定测试语言，全部 列显示使用整个训练集（所有语言）训练时的结果，不含 列显示使用所有不 属于测试语言的训练实例进行训练时的结果，仅 列则显示仅使用测试语言的训练实例进行训练时的结果。例如，当测试语言为 JavaScript 时，不含 列显示使用所有不 属于 JavaScript 的训练数据样本进行训练时的性能，而仅 列显示仅使用 JavaScript 训练样本进行训练时的性能。

在所有编程语言中，在所有编程语言上训练的模型具有更高的合理率，这表明在所有情况下都存在某种知识迁移。我们发现，训练集较小的编程语言（Go、PHP、JavaScript）从跨语言训练中获益最大。相反，对于训练集较大的编程语言（C++、Python），这种提升要小得多（1-2%）。

对于 Java、C语言、PHP 和 JavaScript，使用所有其他编程语言进行训练的结果优于仅使用该语言进行训练。对于 PHP 和 JavaScript，这可能是因为它们的训练集非常小。对于 Java 和 C语言，我们推测这可能是由于它们与 C++ 相似（即语法和类型相似）。以 Java 为例，鉴于这种相似性，尽管存在语言差异，模型从超过 250,000 个 C/C++ 软件缺陷（以及其他非 Java 编程语言）中学到的缺陷修复模式，可能比从 13,000 个 Java 软件缺陷中学到的更多。

> **研究问题2.3的答案**
>
> 我们清楚地观察到不同编程语言之间存在一定的知识迁移。对于训练集较小的语言，这种迁移效应更为显著。这一结果初步表明，数据可用性较低的语言能够从其他语言的训练数据中受益。

5.2.5 NPR中的候选项数量（研究问题2.4）

在使用 NPR 基线进行的实验中，我们为每个缺陷生成了五个候选项。这个数量旨在平衡代码生成的计算成本与数量足够的候选项。然而，一个不可避免的问题是：如果生成更少或更多的候选项，性能会如何变化。为回答这个问题，对于在所有编程语言上训练的模型，

我们最多生成并评估 10 个候选项。图7展示了每个缺陷生成 1 至 10 个候选项时的性能（合理率）。只有一个候选项时，性能降至 46.13%；而有 10 个候选项时，性能提升至 64.33%。然而，将计算成本从 5 个候选项增加到 10 个候选项，即计算成本翻倍，仅带来了 7.5% 的相对性能提升。我们的在线补充材料包含候选项评估结果的图形化展示（见附录中的表6）。

> **对 RQ2.4 的回答**
> 每个缺陷仅生成一个候选项时，总体合理率只有 46%。每个缺陷生成 10 个候选项时，该比例会提高到 64% 以上。不过，随着候选项数量增加，相对提升幅度会逐渐减小。

6 讨论

根据基线的结果，我们可以得出结论：仍有很大的改进空间，而且不同类型的方法面临着不同的挑战。

生成与验证方法面临的挑战。 我们的 G&V 基线 Cardumen 的性能相当令人失望。不过，至少有两个因素需要考虑：时间预算相当有限，而且由于提交中的冗余度较低，修复成分的搜索空间也非常有限。如果放宽这两项限制，性能可能会有所提升。时间限制凸显了 G&V 方法的一个既有问题：这类方法可能非常缓慢，从而妨碍对其进行大规模评估。综合来看，这两个因素表明，若要与 NPR 方法竞争，自动程序修复就需要更高效的搜索方法。G&V 方法面临的第二个挑战是其依赖特定语言：我们只能在基准测试集中的 Java 部分运行 Cardumen。这确实缓解了规模扩展问题，因为测试集的大小缩减了一个数量级。然而，与 NPR 方法能够轻松支持多种语言相比，缺乏语言无关性仍然是一项重大局限。

神经程序修复方法面临的挑战。我们对 NPR 基线的性能感到欣喜。这既表明该基准测试集的难度适中，也说明改进空间很大：我们的直接基线无法修复超过 40% 的软件缺陷（使用 10 个候选项时为 35%）。此外，我们可以预期，剩余的 40% 软件缺陷将更具挑战性。NPR 方法的性能与缺陷修复规模高度相关（远高于 G&V 方法），这表明 NPR 方法能够很好地处理“较简单”的软件缺陷，但对更复杂的软件缺陷效果不佳。模型在最成功和最不成功的标签上的表现也印证了这一点：它最擅长的是较简单的修复，例如运算符使用方面的问题；而最薄弱之处则在于与程序控制流相关的问题。

利用迁移学习。迁移学习带来了令人鼓舞的结果。这些结果解释了为何训练集规模与性能之间的负相关程度低于预期：如果能够从相似的编程语言（或其他编程语言中的相似问题）中学习，那么训练集规模较小就不会造成太大问题；这也为将 NPR 技术应用于不太流行的编程语言开辟了可能性。因此，未来应进一步研究如何定义模型，使其能比我们的基线更有效地利用其他编程语言中的数据。另一方面，迁移学习的效果或许存在上限：对于更复杂的软件缺陷，或涉及更特殊语言特性的软件缺陷（例如 Ruby 中的关键字参数），迁移学习的效果可能较差。

利用执行数据。到目前为止，我们提供的基线性能数据仅利用了静态数据。我们完全没有将收集到的执行数据用作学习信号。我们认为，利用这一信号将显著提升模型性能。只需向模型提供需要修复的运行时错误类型（例如空指针异常），就能提供极其宝贵的线索，因为这有助于缩小问题范围。同样，知道问题并非运行时错误，而是其他问题（例如计算错误），也有助于缩小问题范围。更进一步，了解有关失败测试用例的更多信息，可能对神经程序修复模型非常有价值。

迈向混合方法? 另一种利用执行数据的途径，是向模型提供有关修复尝试的反馈。RewardRepair [85] 等方法是朝这一方向迈出的第一步；然而，RewardRepair 受到包含实际执行信息的训练数据量限制。可以设想这些方法的多种变体，它们可以构成经典生成与验证方法和神经程序修复方法之间的混合方案：神经程序修复方法可以生成补丁，并通过执行来确认其是否可行；如果不可行，无法正常运行。则可以多次调用该神经程序修复方法，直至测试通过。ARJANMT [42]，作为一种生成与验证方法-神经程序修复方法混合方案，

例如，将 NPR 模型用作进一步修复成分的来源。让这类系统进一步利用执行信息，或许能提高其效率。神经程序修复系统还可以通过其他方式扩展，例如向其提供先前执行的数据，也许可以采用类似“草稿板”[55] 的机制，或者允许它们使用工具，例如 Toolformer [66]。

拓展基准测试集。 未来可以拓展 RunBugRun。如果基准测试集上的性能开始趋于饱和（例如超过 95%），进展将变得更难衡量。若出现这种情况，我们认为可以通过以下几种方式提高任务的难度：

- RunBugRun 中的缺陷修复规模从 1 到 6 个词元不等。我们预计，较小变更的进展会更快。更新后的版本可以纳入规模更大的软件缺陷，或者降低最小规模缺陷修复的权重。
- 同样，我们预计有些错误类别会比其他类别更快得到解决。更新后的 RunBugRun 可以过滤掉这些类别，或者提高最具挑战性的类别的权重。
- 第三种可能是增加编程语言的数量，加入较不常用、可供学习的数据较少的语言，以进一步鼓励利用迁移学习的模型。另一种做法是直接限制训练数据中的缺陷数量，以鼓励开发样本高效模型。
- 最后，为了更加重视模型泛化到新问题的能力，可以调整基准测试的拆分方式，使测试集中的问题不出现在训练集中，反之亦然。

7 局限性、偏差与有效性威胁

范围有限 该数据集由针对算法问题的简短解决方案实现组成。数据集中的程序代码与大型软件项目中的代码存在显著差异，后者可能包含复杂的类层次结构、用户界面相关代码、大量方法、函数和代码文件、对软件框架和库的依赖、构建脚本等。尽管这两类代码之间存在一定差异，但根据我们的研究结果，该数据集中的缺陷类型分布与先前研究考察的开源项目中的软件缺陷分布相似 [56]。此外，正如我们之前指出的，简单程序执行效率很高，也便于开展大量实验。而且，大型项目中的许多缺陷对于当前的自动程序修复系统来说过于复杂（尤其是需要多处变更，或涉及多个方法和文件的缺陷）。因此，我们认为，本文介绍的数据集是一个很有意义的测试平台，有助于推动后续重要工作，最终实现修复复杂软件系统中缺陷的目标。

缺乏多样性 测试集包含用于解决 2,693 个不同问题的程序。由于针对同一问题的解题程序不可避免地具有一定相似性，代码的多样性以及由此产生的软件缺陷多样性在一定程度上受到限制。我们通过提供错误标签来缓解这一问题，以便更好地了解不同类型的软件缺陷及其各自的性能。

分词 我们为全部八种编程语言实现了分词器。我们努力按照相应的语言规范，正确地对其中大部分内容进行词法切分。然而，要完全按照规范对这八种编程语言进行分词，需要投入大量工程工作量，远超本研究的范围。

标注 为了标注软件缺陷，我们手动制定了一组规则，用于匹配解析树中的特定模式。由于解析树格式或语言特性的差异，这类规则通常因语言而异。我们尽力覆盖大量变更。然而，我们无法完全排除某一重要类别的变更未能得到充分捕捉的可能性。遗憾的是，现有解决方案（如 GumTree [22] ）目前尚不支持我们数据集中的所有编程语言。随着现有工具不断改进，并逐步支持越来越多的编程语言，我们希望未来能够改用这类更稳健、更先进的工具。

合理性与正确性人工评估补丁正确性既繁琐又低效。PATCH/TEST-SIM [79], DiffTGen [78] 和 RGT [84] 都致力于自动化这项工作。尽管如此，人工评估仍然是常见做法。然而，当测试集规模达到数千时，若要在合理的工作量内完成人工评估已不再可行。

我们数据集中的许多测试都取自用于训练和评估 AlphaCode 的数据集。AlphaCode [44], 是一种深度学习模型，能够根据问题描述合成完整的求解程序。作者估计，有 4% 的

通过测试套件的合成程序实际上并不正确。我们原本预计，看似合理但修复错误的程序所占比例也在相近范围内。为了获得更准确的估计，我们手动评估了 CodeT5 生成的 200 个随机抽取的合理修复样本，发现其中 7 个（3.5%）不正确。样本中有 46.5% 的修复与真实补丁完全相同。对于 19 个补丁，判断其正确性非常困难。在这种情况下，我们使用随机输入运行打补丁后的程序，并将其输出与真实正确程序在相同输入下的输出进行比较。如果能在合理的时间内（1–2 分钟）找到输出不一致的输入，我们就认为该补丁无效；否则则认为有效。我们的评估结果收录于在线补充材料[5]中。以往研究报告的看似合理但错误的修复比例通常更高。例如，对于 CoCoNuT，Lutellier 等人[47] 报告称，QuixBugs 的这一比例为 35%；Lin 等人 [45] 报告称，Codeflaws 的这一比例为 41% [69]。以往研究表明 [84]，手动评估正确性容易出错。如果未来研究发现某些问题的测试套件较弱，可以利用所有程序均可执行这一事实生成额外的测试用例。最后，本研究呈现的数据集也可作为进一步研究自动补丁正确性评估的起点。

8 总结

本文介绍了 RunBugRun，这是一个大型、多语言的可执行缺陷数据集。我们的基线评估表明，仍有改进空间，尤其是生成与验证方法。此外，结果显示，性能因编程语言和所需变更数量的不同而有显著差异。我们的 CodeT5 基线在 Python 上的合理率达到 69%，但在 JavaScript、Go 和 Ruby 上的性能明显较低（44.5%–47.8%）。总体而言，数据可用性较低的语言，其性能也较低。值得庆幸的是，实验表明，低资源语言能够受益于知识迁移效应：混入数据充足的热门语言的训练数据后，我们尝试的所有组合的性能均有所提升。修复缺陷所需的变更数量增加时，性能也会下降。NPR 基线生成的单项变更修复方案中，68.7% 达到了合理标准。然而，对于需要多项变更的缺陷，这一比例会下降：需要四项或更多变更时，只有 49% 的案例生成了合理修复方案；需要六项变更时，这一比例低于 43%。这表明，要让自动程序修复系统能够处理涉及数十项变更和多个代码块的复杂缺陷，我们还有很长的路要走。尽管如此，随着自动程序修复工具不断进步，未来可以提高变更数量上限，以获得更具挑战性的缺陷集合（我们仅纳入了可通过少于七项词元修改修复的缺陷）。

本文仅标志着 RunBugRun 数据集建设的起点。我们计划从多个方面持续扩展该数据集。例如，目前的数据集包含以八种编程语言编写的程序（C语言、C语言++、Python、Java、Ruby、JavaScript、Go、PHP）。未来，我们计划加入更多编程语言。我们尤其期待未来能够利用运行时信息开展工作。由于所有程序都可执行，因此可以轻松获取运行轨迹，并将其用作机器学习模型的唯一输入或补充输入。我们计划构建一个包含此类执行轨迹的并行数据集。图 11 展示了一个从 Python 数据集实例中获取的轨迹示例。尤其是在故障定位方面，这类信息具有巨大潜力。例如，Wang、Singh 和 Su [73] 曾利用执行轨迹的神经嵌入来引导 SARFGEN [74]，（一种用于入门编程练习的自动程序修复工具）的搜索。最近，Gupta 等人 [25] 在一个神经程序合成系统中利用执行轨迹修复了错误合成的程序。

关于如何下载、安装和运行 RunBugRun 的详细信息，请参阅项目网站 https:// github.com/giganticode/run_bug_run。该网站不仅提供数据集本身，还提供可安全执行提交的基础设施。

参考文献

[1] Wasi Uddin Ahmad 等人。面向程序理解与生成的统一预训练。2021年。数字对象唯一标识符：10.48550/ARXIV.2103.06333。URL：https://arxiv.org/abs/2103.06333。[2] Toufique Ahmed 和 Premkumar Devanbu。“面向软件工程的多语言训练”。载于：第*44*届国际软件工程会议论文集。ICSE '22。宾夕法尼亚州匹兹堡：计算机协会，2022年，页码 1443–1455。ISBN：9781450392211。数字对象唯一标识符：10.1145/3510003.3510049。URL：https://doi-org.libproxy.unibz.it/10.1145/3510003.3510049。[3] Miltiadis Allamanis、Henry Jackson-Flux 和 Marc Brockschmidt。“自监督错误检测与修复”。载于：神经信息处理系统进展。第34卷。Curran Associates, Inc.，2021年，页码 27865–27876。URL：https : / / proceedings . neurips . cc / paper / 2021 / hash / ea96efc03b9a050d895110db8c4af057-Abstract.html（访问日期：2022年11月23日）。

[5] https:/ /giganticode.github.io/rbugr/assessm ent/

[4] Hlib Babii 等人。“利用协作式启发式仓库挖掘软件仓库”。载于：*2021*年 *IEEE/ACM* 第*43*届国际软件工程会议：新想法与新兴成果（*ICSE-NIER*）。2021年 IEEE/ACM 第43届国际软件工程会议：新想法与新兴成果（ICSE-NIER）。2021年5月，页码 106–110。数字对象唯一标识符：10.1109/ICSE-NIER52604.2021.00030。[5] Earl T. Barr 等人。“整形手术假说”。载于：第*22*届 *ACM SIGSOFT* 软件工程基础国际研讨会论文集。FSE 2014。中国香港：计算机协会，2014年，页码 306–317。ISBN：9781450330565。数字对象唯一标识符：10.1145/2635868.2635898。URL：https: // doi-org.libproxy.unibz.it/10.1145/2635868.2635898。[6] Emily M. Bender 等人。“论随机鹦鹉的危险：语言模型会不会太大？”载于：*2021*年 *ACM* 公平性、问责制与透明度会议论文集。FAccT ’21。美国纽约州纽约：计算机协会，2021年3月3日，页码 610–623。ISBN：978-1-4503-8309-7。数字对象唯一标识符：10.1145/3442188.3445922。URL：http://doi.org/10.1145/3442188.3445922（访问日期：2022年8月29日）。[7] Gareth Bennett、Tracy Hall 和 David Bowes。“某些自动生成的补丁比其他补丁更可能正确：对 Defects4J 补丁特征的分析”。载于：第三届国际自动程序修复研讨会论文集。2022年，页码 46–52。[8] Samuel Benton、Ali Ghanbari 和 Lingming Zhang。“Defexts：面向现代 Java虚拟机语言的可复现现实世界缺陷精选数据集”。载于：*2019*年 *IEEE/ACM* 第*41*届国际软件工程会议：伴随论文集（*ICSE-Companion*）。2019年 IEEE/ACM 第41届国际软件工程会议：伴随论文集（ICSE-Companion）。2019年5月，页码 47–50。数字对象唯一标识符：10 . 1109/ ICSE - Companion.2019.00035。[9] Tom Britton 等人。可逆调试软件“量化使用可逆调试器节省的时间和成本”。[10] Eduardo C. Campos 和 Marcelo de A. Maia。“发现常见缺陷修复模式：一项大规模观察性研究”。载于：软件：演进与过程31.7（2019），e2173。ISSN：2047-7481。数字对象唯一标识符：10.1002/smr.2173。URL：http://onlinelibrary.wiley.com/doi/abs/10.1002/smr. 2173（访问日期：2021年5月31日）。[11] José Campos 等人。“Gzoltar：用于测试与调试的 Eclipse插件”。载于：第*27*届 *IEEE/ACM* 自动化软件工程国际会议论文集。2012年，页码 378–381。[12] Liushan Chen、Yu Pei 和 Carlo A. Furia。“无需契约的基于契约的程序修复”。载于：*2017*年 *IEEE/ACM* 第*32*届自动化软件工程国际会议（*ASE*）。2017年 IEEE/ACM 第32届自动化软件工程国际会议（ASE）。2017年10月，页码 637–647。数字对象唯一标识符：10 . 1109/ASE.2017.8115674。[13] Zimin Chen 等人。“SequenceR：用于端到端程序修复的序列到序列学习”。2019年9月9日。数字对象唯一标识符：10 . 1109 / TSE . 2019 . 2940179。arXiv：1901 . 01808 [cs, stat]。URL：http : / / arxiv.org/abs/1901.01808（访问日期：2021年2月15日）。[14] Zimin Chen 和 Martin Monperrus。“CodRep 源代码机器学习竞赛 ”。2018 年 11 月 14 日。arXiv：1807 . 03200 [cs]。URL：http : / / arxiv . org / abs / 1807 . 03200（访问日期：2021年3月4日）。[15] Roberta Coelho 等人。“Android 中的异常处理缺陷风险”。载于：实证软件工程22.3（2017），页码 1264–1304。[16] Viktor Csuvik 和 László Vidács。“FixJS：缺陷修复JavaScript提交数据集”。载于：*2022*年 *IEEE/ACM* 第*19*届软件仓库挖掘国际会议（*MSR*）。2022年 IEEE/ACM 第19届软件仓库挖掘国际会议（MSR）。2022年5月，页码 712–716。数字对象唯一标识符：10.1145/3524842. 3528480。[17] Elizabeth Dinella 等人。“HOPPITY：学习图变换以检测并修复程序中的缺陷”。载于：第八届国际学习表征会议。2020年4月。URL：https://iclr.cc/virtual_2020/poster_SJeqs6EFvB.html（访问日期：2021年3月5日）。[18] Dawn Drain 等人。“DeepDebug：利用堆栈跟踪、回译和代码骨架修复 Python 缺陷”。载于：*arXiv*：*2105.09352 [cs]*（2021年5月19日）。arXiv：2105.09352。URL：http://arxiv.org/abs/2105. 09352（访问日期：2021年9月17日）。[19] Thomas Durieux 等人。“Java程序修复工具的实证综述：针对 2,141 个缺陷和 23,551 次修复尝试的大规模实验”。载于：*2019*年第*27*届 *ACM* 欧洲软件工程会议与软件工程基础研讨会联合会议论文集。2019年，页码 302–313。

[20] Thomas Durieux 和 Martin Monperrus。“DynaMoth：用于自动程序修复的动态代码合成”。载于：*2016* 年 *IEEE/ACM* 第 *11* 届软件测试自动化国际研讨会（*AST*）。2016 年 IEEE/ACM 第 11 届软件测试自动化国际研讨会（AST）。2016 年 5 月，第 85–91 页。DOI：10.1109/ AST.2016.021.[21] Thomas Durieux 和 Martin Monperrus。*IntroClassJava*：包含 *297* 个小型缺陷 *Java* 程序的基准测试集。研究报告 hal-01272126。里尔第一大学，2016 年。URL：https://hal.archives- ouvertes. fr/hal-01272126（访问于 2021 年 9 月 18 日）。[22] Jean-Rémy Falleri 等人。“细粒度且准确的源代码差异比较”。载于：第 *29* 届 *ACM/IEEE* 自动化软件工程国际会议论文集。2014 年，第 313–324 页。[23] Stephanie Forrest 等人。“一种用于自动化软件修复的遗传编程方法”。载于：第 *11* 届遗传与进化计算年度会议论文集。GECCO ’09。美国纽约州纽约市：计算机协会，2009 年 7 月 8 日，第 947–954 页。ISBN：978-1-60558-325-9。DOI：10.1145/ 1569901.1570031。URL：http://doi.org/10.1145/1569901.1570031（访问于 2021 年 2 月 22 日）。[24] C. Le Goues 等人。“GenProg：一种用于自动化软件修复的通用方法”。载于：*IEEE* 软件工程汇刊38.1（2012 年 1 月），第 54–72 页。ISSN：1939-3520。DOI：10.1109/TSE.2011.104。[25] Kavi Gupta 等人。“合成、执行与调试：学习为神经程序合成进行修复”。载于：神经信息处理系统进展33（2020 年），第 17685–17695 页。[26] Péter Gyimesi 等人。“BugsJS：JavaScript 缺陷基准测试集”。载于：*2019* 年第 *12* 届 *IEEE* 软件测试、验证与确认会议（*ICST*）。2019 年第 12 届 IEEE 软件测试、验证与确认会议（ICST）。2019 年 4 月，第 90–101 页。DOI：10.1109/ICST.2019.00019。[27] Md Mahim Anjum Haque 等人。*FixEval*：针对编程问题的程序修复执行评估。2022 年 9 月 29 日。DOI：10.48550/arXiv.2206.07796。arXiv：2206.07796 [cs]。URL：http: //arxiv.org/abs/2206.07796（访问于 2022 年 11 月 23 日）。[28] Jingxuan He、Luca Beurer-Kellner 和 Martin Vechev。“基于学习的缺陷检测器中的分布偏移”。载于：第 *39* 届国际机器学习会议论文集。Kamalika Chaudhuri 等人编。第 162 卷。《机器学习研究论文集》。PMLR，2022 年 7 月 17–23 日，第 8559–8580 页。URL：https://proceedings.mlr.press/v162/he22a.html。[29] Kim Herzig 和 Andreas Zeller。“纠缠的代码变更的影响”。载于：*2013* 年第 *10* 届软件仓库挖掘工作会议（*MSR*）。2013 年第 10 届软件仓库挖掘工作会议（MSR）。2013 年 5 月，第 121–130 页。DOI：10.1109/MSR.2013.6624018。[30] Jinru Hua 等人。“SketchFix：一种采用惰性候选项生成的自动程序修复方法工具”。载于：*2018* 年欧洲软件工程会议与软件工程基础研讨会第 *26* 届 *ACM* 联合会议论文集。ESEC/FSE 2018。美国纽约州纽约市：计算机协会，2018 年 10 月 26 日，第 888–891 页。ISBN：978-1-4503-5573-5。DOI：10.1145/3236024. 3264600。URL：http://doi.org/10.1145/3236024.3264600（访问于 2021 年 9 月 3 日）。[31] Hamel Husain 等人。“CodeSearchNet 挑战：评估语义代码搜索的发展现状”。载于：*arXiv* 预印本 *arXiv:1909.09436*（2019 年）。[32] Nan Jiang、Thibaud Lutellier 和 Lin Tan。“CURE：用于自动程序修复的代码感知神经机器翻译”。2021 年 2 月 26 日。arXiv：2103.00073 [cs]。URL：http://arxiv.org/abs/2103. 00073（访问于 2021 年 3 月 22 日）。[33] Pratik Joshi 等人。“自然语言处理领域语言多样性与包容性的现状及未来”。载于：第 *58* 届计算语言学协会年会论文集。ACL 2020。线上：计算语言学协会，2020 年 7 月，第 6282–6293 页。DOI：10.18653/v1/2020.acl-main.560。URL：https://aclanthology.org/2020.acl-main.560（访问于 2022 年 11 月 17 日）。[34] René Just、Darioush Jalali 和 Michael D. Ernst。“Defects4J：现有故障数据库，支持针对 Java 程序开展受控测试研究”。载于：*2014* 年国际软件测试与分析研讨会论文集。ISSTA 2014。美国纽约州纽约市：计算机协会，2014 年 7 月 21 日，第 437–440 页。ISBN：978-1-4503-2645-2。DOI：10.1145/2610384.2628055。URL：http://doi.org/ 10.1145/2610384.2628055（访问于 2021 年 4 月 8 日）。[35] Rafael-Michael Karampatsis 和 Charles Sutton。“单语句缺陷有多常见？ManySStuBs4J 数据集”。载于：第 *17* 届软件仓库挖掘国际会议论文集。MSR ’20。美国纽约州纽约市：计算机协会，2020 年 6 月 29 日，第 573–577 页。ISBN：978-1-4503-7517-7。DOI：10.1145/3379597.3387491。URL：https://doi.org/10.1145/3379597.3387491（访问于 2021 年 4 月 14 日）。

[36] Sunghun Kim 和 E. James Whitehead。“修复缺陷花了多长时间？”载于：*2006* 年软件仓库挖掘国际研讨会论文集。MSR ’06。美国纽约州纽约市：计算机协会，2006 年 5 月 22 日，第 173–174 页。ISBN：978-1-59593-397-3。DOI：10.1145/1137983. 1138027。URL：http://doi.org/10.1145/1137983.1138027（访问于 2021 年 6 月 3 日）。[37] Herb Krasner。“美国低劣软件质量的代价：2020 年报告”。载于：（2020 年），第 46 页。[38] Thomas D. LaToza、Gina Venolia 和 Robert DeLine。“维护心智模型：一项关于开发者工作习惯的研究”。载于：第 *28* 届国际软件工程会议论文集——*ICSE ’06*。第 28 届国际会议论文集。中国上海：ACM 出版社，2006 年，第 492 页。ISBN：978-1-59593-375-1。DOI：10.1145/1134285.1134355。URL：http://portal.acm.org/citation.cfm? doid=1134285.1134355（访问于 2021 年 3 月 15 日）。[39] Hung Le 等人。*CodeRL*：通过预训练模型和深度强化学习掌握代码生成。2022 年。DOI：10.48550/ARXIV.2207.01780。URL：https://arxiv.org/abs/2207. 01780。[40] Xuan-Bach D Le 等人。“补丁正确性评估的可靠性”。载于：*2019* 年 *IEEE/ACM* 第 *41* 届国际软件工程会议（*ICSE*）。IEEE。2019 年，第 524–535 页。[41] Claire Le Goues 等人。“用于 C 程序自动修复研究的 ManyBugs 和 IntroClass 基准测试集”。载于：*IEEE* 软件工程汇刊 41.12（2015 年 12 月），第 1236–1256 页。ISSN：1939-3520。DOI：10.1109/TSE.2015.2454513。[42] Dongcheng Li 等人。“利用神经机器翻译改进基于搜索的自动程序修复”。载于：*IEEE Access*10（2022 年），第 51167–51175 页。DOI：10.1109/ACCESS.2022.3164780。[43] Yi Li 、 Shaohua Wang 和 Tien N. Nguyen。“DLFix：基于上下文的代码转换学习，用于自动程序修复”。载于：*ACM/IEEE* 第 *42* 届国际软件工程会议论文集。ICSE ’20。美国纽约州纽约市：计算机协会，2020 年 6 月 27 日，第 602–614 页。ISBN：978-1-4503-7121-6。DOI：10.1145/3377811.3380345。URL：http://doi.org/10.1145/ 3377811.3380345（访问于 2021 年 4 月 1 日）。[44] Yujia Li 等人。使用 *AlphaCode* 实现竞赛级代码生成。2022 年。DOI：10.48550/ARXIV.2203. 07814。URL：https://arxiv.org/abs/2203.07814。[45] Derrick Lin 等人。“QuixBugs：基于 Quixey 挑战的多语言程序修复基准测试集”。载于：*2017* 年 *ACM SIGPLAN* 系统、编程、语言与软件惠民国际会议附属会议论文集。SPLASH Companion 2017。美国纽约州纽约市：计算机协会，2017 年 10 月 22 日，第 55–56 页。ISBN：978-1-4503-5514-8。DOI：10. 1145 / 3135932.3135941。URL：http: // doi .org / 10. 1145 /3135932 . 3135941（访问于 2021 年 4 月 21 日）。[46] Kui Liu 等人。“自动程序修复系统评估的批判性综述”。载于：*JournalofSystemsand Software* 171（2021 年 1 月），第 110817 页。ISSN：01641212。DOI：10.1016/j.jss.2020.110817。URL：https://linkinghub.elsevier.com/retrieve/pii/S0164121220302156（访问于 2021 年 3 月 23 日）。[47] Thibaud Lutellier 等人。“CoCoNuT：通过集成结合上下文感知神经翻译模型进行程序修复”。载于：第 *29* 届 *ACM SIGSOFT* 国际软件测试与分析研讨会论文集。ISSTA 2020。美国纽约州纽约市：计算机协会，2020 年 7 月 18 日，第 101–114 页。ISBN：978-1-4503-8008-9。DOI：10.1145/3395363.3397369。URL：http://doi.org/10. 1145/3395363.3397369（访问于 2021 年 4 月 1 日）。[48] Fernanda Madeiral 等人。“Bears：用于自动程序修复研究的可扩展 Java 缺陷基准测试集”。载于：*2019* 年 *IEEE* 第 *26* 届国际软件分析、演化与再工程会议（*SANER*）（2019 年 2 月），第 468–478 页。DOI：10.1109/SANER.2019.8667991。arXiv：1901.06024。URL：http: //arxiv.org/abs/1901.06024（访问于 2021 年 9 月 18 日）。[49] Amirabbas Majd 等人。“Code4Bench：用于不同程序分析技术的 Codeforces 数据多维基准测试集”。载于：*Journal of Computer Languages*53（2019 年 8 月 1 日），第 38–52 页。ISSN：2590-1184。DOI：10 . 1016 / j . cola . 2019 . 03 . 006。URL：https : / / www . sciencedirect . com / science/article/pii/S1045926X18302489（访问于 2022 年 11 月 23 日）。[50] Amirabbas Majd 等人。“Code4Bench：用于不同程序分析技术的 Codeforces 数据多维基准测试集”。载于：*Journalof ComputerLanguages* 53（2019 年），第 38–52 页。ISSN：2590-1184。DOI：https://doi.org/10.1016/j.cola.2019.03.006。URL：https://www.sciencedirect.com/science/article/pii/S1045926X18302489。[51] Matias Martinez 和 Martin Monperrus。“Astor：用于 Java 的程序修复库”。载于：第 *25* 届软件测试与分析国际研讨会论文集。2016 年，第 441–444 页。

[52] Matias Martinez 和 Martin Monperrus。《利用自动挖掘模板构建超大型修复搜索空间：Astor 的 Cardumen 模式》。载于《基于搜索的软件工程》。Thelma Elita Colanzi 和 Phil McMinn 编。Cham：Springer International Publishing，2018年，第 65–86 页。ISBN：978-3-319-99241-9。[53] Martin Monperrus 等人。《Megadiff：一个包含 60 万项 Java 源代码变更并按差异大小分类的数据集》。2021年8月10日。arXiv：2108。04631 [cs]。网址：http://arxiv.org/abs/2108.04631（访问于 2021年9月19日）。[54] Kunihiro Noda 等人。《经验报告：自动程序修复对工业软件有多有效？》。载于《2020 IEEE 第27届国际软件分析、演化与再工程会议（SANER）》。2020年2月，第 612–616 页。DOI：10.1109/SANER48275.2020.9054829。[55] Maxwell Nye 等人。《展示你的工作：语言模型中间计算的草稿板》。2021年。DOI：10.48550/ARXIV.2112.00114。网址：https://arxiv.org/abs/2112.00114。[56] Kai Pan、Sunghun Kim 和 E. James Whitehead。《理解缺陷修复模式》。载于《实证软件工程》14.3（2009年6月1日），第 286–315 页。ISSN：1382-3256。DOI：10.1007/s10664-008-9077-5。网址：http://doi.org/10.1007/s10664-008-9077-5（访问于 2021年6月8日）。[57] Ruchir Puri 等人。《CodeNet：用于学习多种编码任务的大规模代码人工智能数据集》。2021年8月29日。DOI：10.48550/arXiv.2105.12655。arXiv：2105。12655 [cs]。网址：http://arxiv.org/abs/2105.12655（访问于 2022年11月16日）。[58] Zichao Qi 等人。《对生成与验证式补丁生成系统中补丁合理性和正确性的分析》。载于《2015年国际软件测试与分析研讨会论文集》。ISSTA 2015。美国纽约：计算机协会，2015年7月13日，第 24–36 页。ISBN：978-1-4503-3620-8。DOI：10.1145/2771783.2771791。网址：http://doi.org/10.1145/2771783.2771791（访问于 2021年3月31日）。[59] Yihao Qin 等人。《不稳定测试对自动程序修复的影响》。载于《2021 IEEE 国际软件分析、演化与再工程会议（SANER）》。2021年，第 295–306 页。DOI：10.1109/SANER50967.2021.00035。[60] Shuo Ren 等人。《CodeBLEU：一种代码合成的自动评估方法》。2020年。DOI：10.48550/ARXIV.2009.10297。网址：https://arxiv.org/abs/2009.10297。[61] Cedric Richter 和 Heike Wehrheim。《我们能从开发者错误中学习吗？从真实缺陷修复中学习定位和修复真实缺陷》。2022年7月1日。DOI：10.48550/arXiv.2207.00301。arXiv：2207.00301[cs]。网址：http://arxiv.org/abs/2207.00301（访问于 2022年8月25日）。[62] Cedric Richter 和 Heike Wehrheim。《TSSB-3M：大规模挖掘单语句缺陷》。载于《2022 IEEE/ACM 第19届国际软件仓库挖掘会议（MSR）》。2022年5月，第 418–422 页。DOI：10.1145/3524842.3528505。[63] Sebastian Ruder。《为什么自然语言处理不应局限于英语》。http://ruder.io/nlp-beyond-english。2020年。[64] Ripon Saha 等人。《Bugs.Jar：一个大规模、多样化的真实世界 Java 缺陷数据集》。载于《2018 IEEE/ACM 第15届国际软件仓库挖掘会议（MSR）》。2018年5月，第 10–13 页。[65] Ripon K Saha 等人。《Elixir：有效的面向对象程序修复》。载于《2017 IEEE/ACM 第32届自动化软件工程国际会议（ASE）》。IEEE，2017年，第 648–659 页。[66] Timo Schick 等人。《Toolformer：语言模型可以自学使用工具》。载于 arXiv 预印本 arXiv:2302.04761（2023年）。[67] André Silva 等人。《FLACOCO：基于工业级覆盖率的 Java 故障定位》。2021年。DOI：10.48550/ARXIV.2111.12513。网址：https://arxiv.org/abs/2111.12513。[68] Edward K. Smith 等人。《治愈是否比疾病更糟？自动程序修复中的过拟合》。载于《2015年第10届软件工程基础联合会议论文集》。ESEC/FSE 2015。美国纽约：计算机协会，2015年8月30日，第 532–543 页。ISBN：978-1-4503-3675-8。DOI：10.1145/2786805.2786825。网址：http://doi.org/10.1145/2786805.2786825（访问于 2021年2月18日）。[69] Shin Hwei Tan 等人。《Codeflaws：用于评估自动程序修复工具的程序设计竞赛基准测试集》。载于《2017 IEEE/ACM 第39届国际软件工程会议伴随会议（ICSE-C）》。2017年5月，第 180–182 页。DOI：10.1109/ICSE-C.2017.76。

[70] David A. Tomassi 等人。《BugSwarm：挖掘并持续扩展可复现的故障与修复数据集》。载于第*41*届国际软件工程会议论文集。ICSE '19。加拿大魁北克省蒙特利尔：IEEE Press，2019年5月25日，第 339–349 页。DOI：10.1109/ICSE.2019.00048。网址：http://doi.org/10.1109/ICSE.2019.00048（访问于 2021年9月20日）。[71] M. Tufano 等人。《通过神经机器翻译学习实际环境中的缺陷修复补丁：一项实证调查》。载于*2018*年第*33*届 *IEEE/ACM* 自动化软件工程国际会议（*ASE*）。2018年9月，第 832–837 页。DOI：10.1145/3238147.3240732。[72] Michele Tufano 等人。《通过神经机器翻译学习实际环境中的缺陷修复补丁：一项实证研究》。2019年5月20日。arXiv：1812。08693 [cs]。网址：http://arxiv.org/abs/1812.08693（访问于 2021年3月4日）。[73] Ke Wang、Rishabh Singh 和 Zhendong Su。《用于程序修复的动态神经程序嵌入》。载于：*arXiv* 预印本 *arXiv:1711.07163*（2017年）。[74] Ke Wang、Rishabh Singh 和 Zhendong Su。《搜索、对齐与修复：入门编程练习的数据驱动反馈生成》。载于*SIGPLAN Not.* 53.4（2018年6月），第 481–495 页。ISSN：0362-1340。DOI：10.1145/3296979.3192384。网址：https://doi-org.libproxy.unibz.it/10.1145/3296979.3192384。[75] Yue Wang 等人。*CodeT5*：用于代码理解与生成的标识符感知统一预训练编码器-解码器模型。2021年。DOI：codet5。网址：https://arxiv.org/abs/2109.00859。[76] Ratnadira Widyasari 等人。《BugsInPy：一个 Python 程序中现有缺陷的数据库，用于支持受控测试与调试研究》。载于第*28*届 *ACM* 欧洲软件工程会议与软件工程基础研讨会联合会议论文集。ESEC/FSE 2020。美国纽约：计算机协会，2020年11月8日，第 1556–1560 页。ISBN：978-1-4503-7043-1。DOI：10.1145/3368089.3417943。网址：http://doi.org/10.1145/3368089.3417943（访问于 2021年9月20日）。[77] Thomas Wolf 等人。*HuggingFace* 的 *Transformers*：最先进的自然语言处理。2019年。DOI：10.48550/ARXIV.1910.03771。网址：https://arxiv.org/abs/1910.03771。[78] Qi Xin 和 Steven P Reiss。《通过测试用例生成识别测试套件过拟合补丁》。载于第*26*届 *ACM SIGSOFT* 国际软件测试与分析研讨会论文集。2017年，第 226–236 页。[79] Yingfei Xiong 等人。《在基于测试的程序修复中识别补丁正确性》。载于第*40*届国际软件工程会议论文集。2018年，第 789–799 页。[80] Deheng Yang 等人。《Defects4j 缺陷的修复成分在哪里？》。载于实证软件工程26.6（2021年9月10日），第 122 页。ISSN：1573-7616。DOI：10.1007/s10664-021-10003-7。网址：https://doi.org/10.1007/s10664-021-10003-7（访问于 2022年11月26日）。[81] He Ye 等人。《在 QuixBugs 基准测试集上对自动程序修复进行全面研究》。载于*2019 IEEE* 第*1*届智能缺陷修复国际研讨会（*IBF*）。2019年，第 1–10 页。DOI：10.1109/IBF.2019.8665475。[82] He Ye 等人。《在 QuixBugs 基准测试集上对自动程序修复进行全面研究》。载于*Journalof Systems and Software* 171（2021年），第 110825 页。ISSN：0164-1212。DOI：https://doi.org/10.1016/j.jss.2020.110825。网址：https://www.sciencedirect.com/science/article/pii/S0164121220302193。[83] He Ye 等人。*SelfAPR*：结合测试执行诊断的自监督程序修复。2022年4月4日。DOI：10.48550/arXiv.2203.12755。arXiv：2203。12755 [cs]。网址：http://arxiv.org/2203.12755（访问于 2022年9月4日）。[84] He Ye、Matias Martinez 和 Martin Monperrus。《面向大规模程序修复的自动补丁评估》。载于实证软件工程 26（2021年），第 1–38 页。[85] He Ye、Matias Martinez 和 Martin Monperrus。《基于执行的反向传播神经程序修复》。载于第*44*届国际软件工程会议论文集。ICSE '22。美国纽约：计算机协会，2022年5月21日，第 1506–1518 页。ISBN：978-1-4503-9221-1。DOI：10.1145/3510003.3510222。网址：http://doi.org/10.1145/3510003.3510222（访问于 2022年9月4日）。[86] Yuan Yuan 和 Wolfgang Banzhaf。《ARJA：通过多目标遗传编程自动修复 Java 程序》。载于*IEEE* 软件工程汇刊46.10（2020年10月），第 1040–1067 页。ISSN：1939-3520。DOI：10.1109/TSE.2018.2874648。[87] Hao Zhong 和 Zhendong Su。《真实缺陷修复的实证研究》。载于*2015 IEEE/ACM* 第*37*届 *IEEE* 国际软件工程会议。第1卷。IEEE，2015年，第 913–923 页。

[88] 钟文康等人。“StandUp4NPR：为神经程序修复系统的实证比较制定标准化设置”。载于：自动化软件工程会议（*ASE'22*）。2022年。[89] Barret Zoph 等人。“面向低资源神经机器翻译的迁移学习”。载于：*2016*年自然语言处理实证方法会议论文集。美国德克萨斯州奥斯汀：计算语言学协会，2016年11月，第1568–1575页。数字对象唯一标识符：10 . 18653 / v1 / D16 - 1163。网址：https :// / aclanthology.org/D16-1163。

更多表格和图示

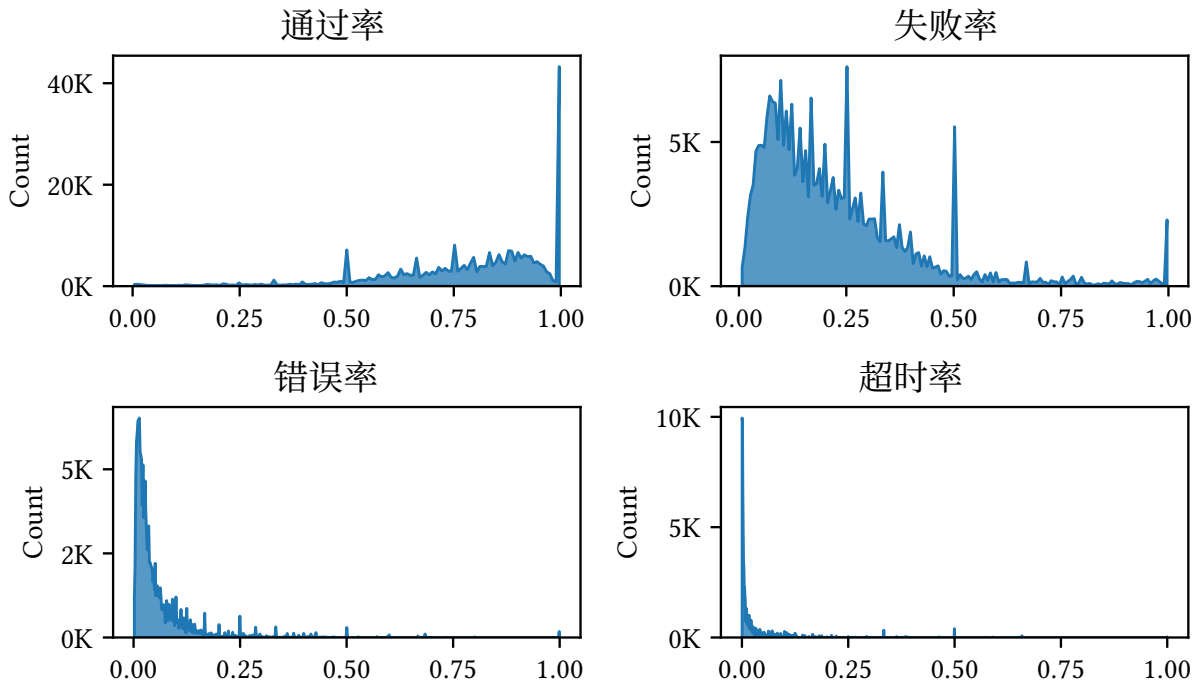


图8：测试用例通过率、失败率、错误率和超时率的分布（测试用例筛选前）。

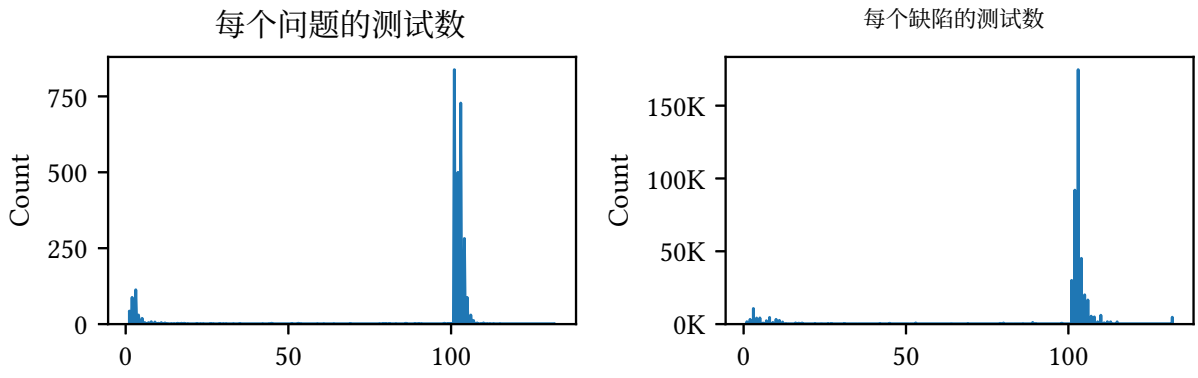


图 9：最终数据集中每个缺陷及问题的测试数分布。

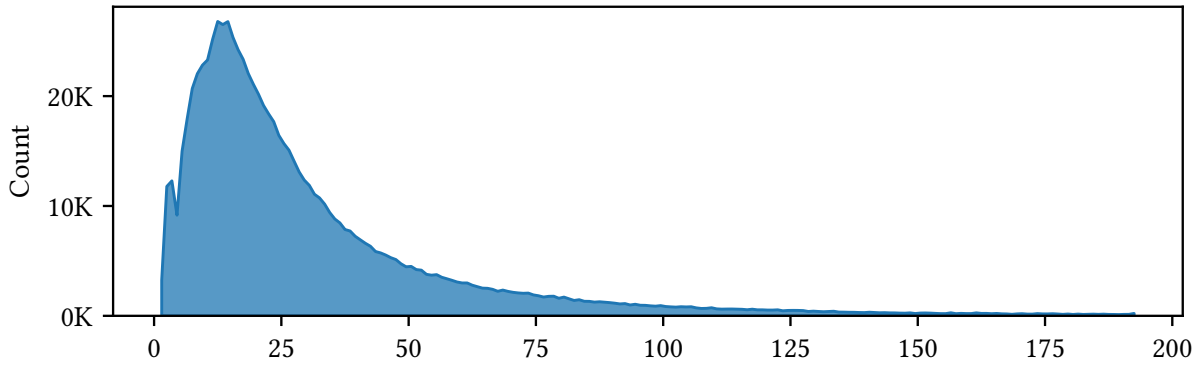
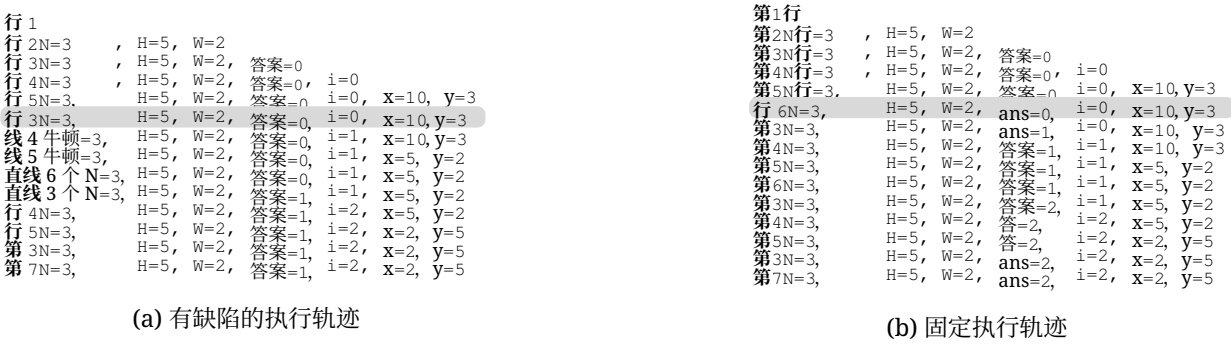


图10：最终数据集中提交程序行数的分布。为便于展示，图中省略了行数最多的1.3%的提交（离群值）。



(a) 有缺陷的执行轨迹

(b) 固定执行轨迹

```
1 N,H,W=映射(整数,输入().拆分())
2 答案=0
3 对于 i 在范围(N):
4
5     如果 x<H 并且y<=
6         答案+=1
7 打印(答案)
```

(c) 补丁

图 11：测试集中错误 #579877 的执行轨迹。左侧为有缺陷版本，右侧为修复后的版本。该缺陷由第5行（底部）错误的 if 条件引起。第5行之后，两条执行轨迹出现分歧（以灰色突出显示）。

表6：候选项评估可视化链接。用户可以按标签筛选评估结果。每行代表给定语言中的一个缺陷，10列分别代表一个候选项。颜色表示评估结果。将鼠标悬停在单元格上可查看详细信息。

C语言 https://giganticode.github.io/rbugr/candidate_vis/c.html C语言++ https://giganticode.github.io/rbugr/candidate_vis/cpp.html Java https://giganticode.github.io/rbugr/candidate_vis/java.html Python https://giganticode.github.io/rbugr/candidate_vis/python.html Ruby https://giganticode.github.io/rbugr/candidate_vis/ruby.html Go https://giganticode.github.io/rbugr/candidate_vis/go.html PHP https://giganticode.github.io/rbugr/candidate_vis/php.html JavaScript https://giganticode.github.io/rbugr/candidate_vis/javascript.html

```
i , 最大值 = gets.split.map(&:to_i)
数组 = gets.split.map(&:to_i).sort
计数 = 0
(01..i-1).each do |n|
  如果 最大值 <数组[n]
    休息
  end
  如果 n ==array.length-1 && max > array[n]
    中断
  end
  max -= 数组[n]
    计数计数 += 1
  end
  puts cnt
```

ID	609453		
语言	Ruby		
拆分	训练		
标签	整数文字变化, 标识符变更		
失败测试	输入	预期输出 6 ...	实际输出 错误 ...
通过测试	输入	预期输出 0 ...	实际输出 0\n ...
错误消息	file.rb:12:in 'block in <Main>': nil:NilClass 的 undefined 方法 '+' （无方法错误） 来自 file.rb:4:在 'each' 中 来自 file.rb:4:在 '<Main>'		

图 12：Ruby 编程语言中的训练样例。提交的程序使用了错误的循环边界和错误的变量，导致抛出异常。原文中的空行已删除，并相应更新了错误消息中的行号。

```
导入 java.util.Scanner;
类 Main{
  公共静态无效 main(String[] args){
    Scanner sc = 新建Scanner(标准输入);
      整数 W =sc.nextInt();
      整数 H = sc.nextInt();
      int x = sc.nextInt();
      int y = sc.nextInt();
      int r = sc.nextInt();
      if ((x +- r) < 0 || W < (x + r) ||
          (y +- r) < 0 || H < (y + r)) {
        System.out.println("否");
        return;
      }
      System.out.println("是");
  }
}
```

ID	112714		
语言	Java		
拆分	Test		
标签	杂项.反义项, 表达式.算术运算符.变化, 控制_flow.branch.if.condition.change		
失败测试	输入	预期输出 Yes ...	实际输出 No ...
通过测试	输入	预期输出 Yes ...	实际输出 Yes ...
错误消息	None		

图13：Java 编程语言中的测试样例。提交的代码包含一个有缺陷的 if 条件（运算符错误）。

vs label count.