北京大学

硕士研究生学位论文

题目：测试文档

姓　　名：王然

学　　号：1501214409

院　　系：信息科学技术学院

专　　业：计算机软件与理论

研究方向：某某方向

导　　师：某某教授

二零一八年五月

版权声明

摘要

程序调试是一个耗费时间的任务，已经有很多研究者提出了各种不同的自动缺陷定位技术去减轻手动调试的负担。在这些自动缺陷定位技术中，基于频谱的缺陷定位和基于状态覆盖的缺陷定位是两中比较常用的缺陷定位技术。这两种技术都是在执行时收集一些统计信息。这两种技术一些相似之处，但是两种技术一直单独地发展，而它们的结合也一直没有被系统地讨论过。

本文对基于频谱的缺陷定位和基于状态覆盖的缺陷定位技术的结合进行了系统的实证研究。首先，本文构建了一个两种技术的统一模型，并在这个模型上系统地探索了四种变体：不同粒度的数据收集、不同的怀疑度公式、不同的怀疑度结合方式、不同的谓词。然后，本文还提出了一个基于机器学习的谓词预测模型，来替代基于状态覆盖的缺陷定位原有的预定义谓词。

本文的研究得到了很多结论。第一，更细粒度的数据收集的效果远远好于粗粒度的数据收集，并且只需要花费稍微多一点的执行时间。第二，把基于频谱的缺陷定位公式应用在基于状态覆盖的缺陷定位的预定义谓词上，其效果反而好于原有的基于状态覆盖的缺陷定位的公式。第三，一个基于频谱的缺陷定位和基于状态覆盖的缺陷定位的线性结合模型，效果比两者都更好。第四，结合方法的效果大部分得益于分支谓词。

关键词：其一，其二

Test Document

Ran Wang (Computer Software and Theory)
Directed by Prof. Somebody

ABSTRACT

Program debugging is a time-consuming task, and researchers have proposed different kinds of automatic fault localization techniques to mitigate the burden of manual debugging. Among these techniques, two popular families are spectrum-based fault localization and statistical debugging, both localizing faults by collecting statistical information at runtime. Though the ideas are similar, the two families have been developed independently and their combinations have not been systematically explored.

In this paper we perform a systematical empirical study on the combination of spectrum-based fault localization and statistical debugging. We first build a unified model of the two techniques, and systematically explores four types of variations: different granularities of data collection, different risk evaluation formulas, different ways of combining suspiciousness scores, and different predicates. Then we propose a machine-learning model to predict the predicates, instead of using pre-defined predicates in statistical debugging.

The study leads to several findings. First, fine-grained data collection significantly outperforms coarse-grained data collection with a little more execution overhead. Second, the risk evaluation formulas of spectrum-based fault localization siginificantly outperforms that of statistical debugging when used in statistical debugging. Third, a linear combination of spectrum-based fault localization and statistical debugging outperforms both individual approaches. Forth, most of the effectiveness of the combined approach contributed by a simple type of predicates: branch conditions.

KEYWORDS: First, Second

目录

第一章　引言

本章主要介绍了自动缺陷定位的研究背景及其重要意义，然后阐述了本文的研究内容、主要贡献和论文结构。

1.1　研究背景

随着软件的发展，生活中越来越多的方面都与软件有着紧密的关系。小到人们的日常出行、购物、餐饮等，大到航空航天、医药等领域，软件在人们的生活中扮演着重要的角色。随着软件的应用领域的扩大，软件的复杂性上升，提升了软件缺陷的可能性。软件缺陷可能会导致巨大的损失。一个著名的被广泛引用的例子是海湾战争时，一颗导弹由于导航软件的精度缺陷而偏离了目标，导致28人死亡和100人受伤[57]。2002年，美国国家标准与技术研究院(NIST)发表的一篇报告[38]显示，软件缺陷每年会导致约595亿美元的经济损失。发现并修复软件缺陷，保障软件的高质量成为一项重要的任务。

1.2　研究意义

在发现软件缺陷之后，开发人员为了解决这个缺陷往往需要三步[35]。第一步，缺陷定位，需要找到程序中和这个缺陷有关的语句。第二步，理解缺陷，明白为什么会发生缺陷。第三步，修复缺陷，修改代码以让缺陷消失。这三个步骤合起来就是调试的过程。缺陷定位作为调试的第一步，其完成速度和准确性对后面的步骤有着很大的影响。在传统的开发环境当中，人们可以手动调试来定位缺陷，比如插入断点、打印日志信息等等。在1989年Collofello等人就指出尝试去减少软件中的错误会花费50%到80%的开发和维护的精力[10]。随着软件的复杂性的上升，手动地定位软件缺陷将会耗费更多开发者的时间和精力。为了提高定位缺陷的速度，研究人员对自动化的缺陷定位展开了研究，并取得了巨大的进展{TODO:cite}。然而在2011年，Partin和Osro的一篇调查[35]通过研究缺陷定位技术在实际应用场景下的效果，发现以往的评价指标并不能准确的反映缺陷定位技术在实际应用中的效果。以往的缺陷定位技术是基于一系列关于开发人员会如何调试的假设，而这些假设在实际场景的某些情况下会失效。自动化缺陷定位技术还有很大的发展空间。

1.3 本文研究内容和主要贡献

为了能提升调试的效率，本文对自动化缺陷定位技术进行了深入的研究。通过在实际缺陷中分析传统的缺陷定位技术的效果，本文提出了一种结合基于频谱的缺陷定位和基于状态覆盖的缺陷定位的方式。本文探索了各种不同的结合方式的效果，分析了基于状态覆盖的缺陷定位公式的不足，并使用基于频谱的缺陷定位公式与其互补。同时，由于基于状态覆盖的缺陷定位使用的预定义谓词的不灵活性，本文提出了一种基于机器学习的谓词预测模型，来替代传统的基于状态覆盖的缺陷定位所使用的预定义谓词。

本文的贡献如下：

- 在实际缺陷中深入分析了基于频谱的缺陷定位的效果，发现了基于频谱的缺陷定位利用的频谱信息粒度不够细，导致缺陷和非缺陷无法区别。
- 在实际缺陷中深入分析了基于状态覆盖的缺陷定位的效果，发现了其怀疑度公式在实际缺陷中并不适用。
- 提出了结合基于频谱的缺陷定位和基于状态覆盖的缺陷定位的方式。基于状态覆盖的缺陷定位的信息粒度比基于频谱的缺陷定位的信息粒度细，而基于频谱的缺陷定位公式在实际缺陷中仍然表现良好，两者结合之后获得了更好的效果。
- 利用结合后的模型，在实际缺陷中分析了基于状态覆盖的缺陷定位的谓词起作用的原因，发现分支是起最大作用的谓词。
- 提出了一种基于机器学习的预测谓词的模型，能够根据语句上下文预测谓词，从而更好地定位缺陷。

1.4 论文结构

本文共七章，结构如下：

第一章为引言，介绍了本文的研究背景、研究意义、研究内容和主要贡献。

第二章为相关工作，介绍了国内外相关领域的研究现状，包括自动缺陷定位技术、机器学习技术和实验数据集三部分。

第三章为问题分析，在实际缺陷中分析了现有自动缺陷定位技术的优势和不足。

第四章介绍了本文提出的结合基于频谱的缺陷定位和基于状态覆盖的缺陷定位的方法，以及基于机器学习的谓词预测模型。

第五章介绍了结合基于频谱的缺陷定位和基于状态覆盖的缺陷定位的方法的实现，以及基于机器学习的谓词预测模型的实现。

第六章是本文提出的方法的实验效果与验证。

第七章是对全文的总结和对未来工作的展望。

第二章　相关工作

本章讲介绍自动缺陷定位、机器学习的相关工作和缺陷定位所使用的数据集以帮助更好理解本文的工作。

2.1　自动缺陷定位相关工作

程序切片[42, 43]是自动调试最早的技术之一，但是程序切片之后可能出错的语句数量仍然比较庞大。为了解决程序切片调试方法的短板，一种通过观察错误程序的执行特征和正确程序的执行特征的调试技术被提出。这些技术通过收集程序执行信息，观察不同的某种特征，来定位缺陷。比如使用路径概要[39]，反例[6, 14]，语句覆盖[19]和谓词值[26, 28]等等。

本文根据北京大学熊英飞研究员对缺陷定位的分类[50]，将缺陷定位分为以下几类。

- 基于切片的缺陷定位
- 基于频谱的缺陷定位
- 基于状态覆盖的缺陷定位
- 基于变异的缺陷定位
- 基于构造正确执行状态的缺陷定位
- 基于算法式调试的缺陷定位
- 基于差异化调试的缺陷定位

本文的研究内容主要根据基于频谱的缺陷定位和基于状态覆盖的缺陷定位。

2.1.1　基于切片的缺陷定位

Weiser在1981年提出的程序切片[42, 43]是自动调试（特别是缺陷定位）最早的技术之一。给定一个程序P和一个在P的语句s中使用的变量v，程序切片会找到P中所有可能会影响s中v的值的语句。如果s中v的值是错误的，那么导致这个错误的缺陷语句一定在这个切片当中。也就是说，不在这个切片当中的语句可以在调试过程中被忽略。尽管程序切片已经减少了可能出错的语句的数量，但是切片中的语句的数量仍然比较大。为了解决这个问题，Korel和Laski在1988年提出了动态程序切片[24]。动态程序切片计算某一个特定执行的切片。后来又有很多的动态程序切片的变种被提出[12, 15, 55, 56]，用于解决调试问题，并且产生了大量研究工作[4, 5, 20, 22, 29, 30, 48]。

2.1.2 基于频谱的缺陷定位

基于频谱的缺陷定位是使用最广泛的自动化缺陷定位方法[50]。程序频谱（Program Spectrum）最早由Reps等人于1997年提出[39]，用于解决千年虫问题。 Harrold 等人在2002年[16]提出使用测试覆盖信息作为频谱信息的调试方法。 Renieris等人在2003年提出使用通过的测试用例和失败的测试用例进行缺陷定位[37]，奠定了此后基于频谱的缺陷定位的基础。

考虑一种极端的情况。比如当某一个语句s被执行的之后，测试用例就会失败。而通过的测试用例都不会执行语句s。那么语句s很有可能就是导致缺陷的语句。找出所有这样的语句s就可以大幅减少需要排查错误的语句。但是，在实际的代码中这种极端的情况很少出现。对于一个出错的语句s，它很可能既被失败的测试用例执行，也被通过的测试用例执行。因为一个语句在其不同的上下文作用下会产生不同的效果。简单地计算通过的测试用例覆盖的语句和失败的测试用例覆盖的语句的差集是无法准确找出错误语句的。利用通过的测试用例覆盖的语句的交集和并集，与失败的测试用例覆盖的语句取差集，是最早的一种基于频谱的缺陷定位方法[37]。这种方法也隐含着基于频谱的缺陷定位的假设：被失败的测试用例执行的语句，更有可能有错误。而被通过的测试用例执行的语句，更有可能是正确的。

Jones等人提出的Tarantula[19]，直观地给开发者展示了每个语句在通过的测试用例和失败的测试用例下的参与情况。参与情况也被称为怀疑度。怀疑度更高的语句会在怀疑列表更靠前的位置。相比于交集并集差集的方法，Tarantula在Siemens数据集上可以将错误的语句放在怀疑列表更前面的位置[18]。

Tarantula之后，又有很多计算怀疑度的公式被提出。效果比较好的Ochiai由Abreu等人提出[1]。 Ochiai由[31]提出用于计算基因的相似度。 Abreu等人将其引入用于计算怀疑度，并与Jaccard[9]，Tarantula，AMPLE[11]比较，发现Ochiai计算的怀疑度使得定位效果更好[1, 2]。

Xie等人在理论上证明了不存在单一最佳公式[49]。

除了直接提出用于计算的公式之外，研究人员也开始使用机器学习的方法去学习怀疑度的公式。 Wong等人提出使用反向传播神经网络来定位缺陷[47]。使用的输入数据是频谱信息（语句覆盖信息）和对应的测试用例是通过还是失败。输入数据每一行对应一个测试用例。第i列为1表示的是该测试用例覆盖了第i个语句，为0则表示没有覆盖。预测的标签为1表示该测试用例失败了，为0表示通过了。为了减少需要分析的可能出错的语句的个数（每一行输入数据的维度），优先使用所有失败的测试用例覆盖的语句。此后Wong又提出了使用径向基核函数的神经网络来定位缺陷[45]。

2.1.3　基于状态覆盖的缺陷定位

在缺陷定位的时候，定位的程序元素的大小也会影响结果。程序元素可以是一条语句，一个方法，一个文件。程序元素的粒度越细，对测试信息的利用越精确。然而单个元素上覆盖的测试数量越少，统计显著性越低。如果把程序的每个执行状态作为程序元素，那么这会是一个比语句更加精细的粒度。定位结果也将更加精细，对测试的利用也会更加充分。但是，几乎不会有两个测试覆盖完全相同的状态，因为一个状态所包含的上下文信息往往十分复杂，很难完全一致。于是使用抽象状态代替具体状态。使用谓词将具体状态划分为抽象状态。谓词是形如`a > 0`这样的条件表达式。

Liblit等人最早提出了预定义谓词来划分状态[26]，并提出了统计性调试。通过预定义在哪些代码结构中插入哪些谓词，统计性调试能够收集到许多抽象状态的覆盖情况。

Liu等人改进了计算公式，提出了SOBER[27]。虽然Liblit的方法可以有效定位一些错误，但是Liblit的方法只考虑了一个谓词是否在一次执行中为真，而没有考虑为真的次数。 SOBER提出新的计算公式，从概率分布的角度来计算怀疑度。

除了预定义谓词以外，研究人员还提出各种从程序中获取谓词的方法。 Le等人提出 Savant[25]，使用程序中的不变式的变化来划分状态。程序中的不变式使用 Daikon[13]挖掘。 Savant使用Learning-to-rank方法，通过分析经典的怀疑度分数和在通过的测试用例和失败的测试用例上观察到的不变式，来定位错误的方法。 Savant基于三个出发点。一，在失败的测试用例和通过的测试用例中表现出不同的不变式的程序元素，被怀疑是有错误的。二，如果这些程序元素拥有很高的经典的怀疑度分数，那么它们更有可能是错误的。三，有一些不变式比其他不变式更加可疑，比如 `x == null`。

2.1.4　基于变异的缺陷定位

变异是对程序的任意随机修改，由变异算子得到。变异分析是测试领域的一个概念，被用于衡量一个测试集的好坏。变异分析在程序中插入变异，得到很多变异体，然后使用一组测试去执行变异体。如果一个测试集中任意测试在一个变异体上得到不同的结果，那么这个变异体被这个测试杀死。能杀死越多变异体的测试集越好。

变异被引入缺陷定位，用于定位缺陷。 Papadakis等人提出Metallaxis[34]，一个基于变异的缺陷定位。 Metallaxis基于两个假设：

- 当变异和错误在一个程序的同一条语句上时，失败的测试用例输出发生变化的概率大于通过的测试用例输出发生变化的概率。
- 当变异和错误不在同一条语句上时，通过测试用例输出发生变化的概率大于失败的测试用例输出发生变化的概率。

m	变异体
m_f	变异m导致输出发生变化的失败的测试用例个数
m_p	变异m导致输出发生变化的通过的测试用例个数
m_{f2p}	变异m导致失败的测试用例变成通过的测试用例的个数
m_{p2f}	变异m导致通过的测试用例变成失败的测试用例的个数
F	失败的测试用例的个数

表 2.1　基于变异的缺陷定位的数学符号及其意义

基于表2.1，Metallaxis的怀疑度计算公式为

$$\text{Metallaxis}(m) = \frac{m_f}{\sqrt{F \times (m_f + m_p)}}$$

与Ochiai类似。

Moon等人提出另一个基于变异的缺陷定位技术MUSE[32]。 MUSE利用变异分析去捕捉单个语句和观察到的缺陷之间的关系。 MUSE基于的两个假设是：

- 一个失败的测试用例，比起在变异了正确语句的变异体上，在变异了错误语句的变异体上更容易变成通过的。
- 一个通过的测试用例，比如在变异了失败语句的变异体上，在变异了正确语句的变异体上更容易变成失败的。

基于表2.1，MUSE的怀疑度计算公式为

$$\text{MUSE}(m) = m_{f2p} - m_{p2f} \times \frac{\sum_m m_{f2p}}{\sum_m m_{p2f}}$$

2.1.5　基于构造正确执行状态的缺陷定位

MUSE通过变异体，可以把失败的测试用例变成通过，通过的测试用例变成失败的。假如有一个变异体，它可以把失败的测试用例变成通过的，且不会影响通过的测试用例，那么这个变异体很可能就是缺陷的补丁。但是直接分析出这样的变异体是很困难的。

Zhang提出的谓词翻转[54]巧妙地避免了直接分析出正确的补丁，而是使用改变程序状态来达到相同的目的。假如出错的是一个布尔表达式，改变程序中一个布尔表达式的取值（把真变成假，或者把假变成真），强制改变执行的分支。假如谓词翻转后，失败的测试用例变成通过的，那么对应的布尔表达式很可能有错误。

谓词翻转是局限在布尔表达式，天使调试[8]则试图解决任意表达式的错误。天使调试要求同时具有天使性和灵活性。天使性是指，存在常量c（天使值）把表达式的求

值结果替换成c，失败的测试变得通过。灵活性是指，对于所有通过的测试中的每一次表达式求值，都可以把求值结果换成一个不同的值，并且测试仍然通过。利用符号执行约束求解计算得到天使值。也由于符号执行的开销，天使调试无法应用到大型程序上。

2.1.6　基于算法式调试的缺陷定位

Shapiro提出的算法式调试[40]，通过对子问题询问“是”或“否”来定位缺陷。算法式调试把复杂的计算步骤拆为小的子问题。算法式调试的一个问题是，子问题的正确结果可能是不知道的。如果是让人进行交互式地判断，那么人需要花费时间计算判断子问题的结果。

2.1.7　基于差异化调试的缺陷定位

差异化调试由Zeller等人提出[52, 53]。不同于以往的使用动态分析或静态分析的方法去关注源代码，差异化调试关注程序状态，特别地，差异化调试关注当程序没有出错时的程序状态和程序出错时的程序状态。差异化调试尝试找到一个最小的修改集合，当把这个集合应用到没有出错时的程序状态后，程序出错了。

2.2　机器学习相关工作

{TODO:机器学习相关工作}

神经网络的定义多种多样，采用 Kohonen 1998年在期刊《Neural Networks》上的定义，为“神经网络是由具有适应性的简单单元组成的广泛并行互连的网络，它的组织能够模拟生物神经系统对真是世界物体所作出的交互反应”。

2.3　缺陷定位的数据集

要研究缺陷定位，需要一个包含缺陷的数据集。这个数据集一般来说，需要有多个缺陷。对每一个缺陷，会有对应的测试用例，和对应的一个正确的版本。这些测试用例中既有通过的，也必定有失败的。失败的这个测试用例就是由缺陷导致。

Siemens数据集[17]是一个很早的数据集，用于测试充分性的实验。它由七个C程序组成，大小在141行到512行之间。这七个C程序衍生出132个有缺陷的C程序。每一个错误版本会恰好有一个缺陷。这个缺陷可能涉及多行甚至多个文件。但是这些程序的缺陷是由作者手动插入的，根据作者的描述其实和一个简单的变异操作非常相似。Siemens数据集的输入被构造用于实现完全的代码覆盖。尽管它一开始并不是被用于缺

陷定位，但是很多缺陷定位技术都使用它来验证效果，比如最早的基于频谱的缺陷定位方法[37]，Ochiai[1, 2]， SOBER[27]，BPNN[47]等等。

Defects4j数据集[21]是一个真实、独立、可重现缺陷的数据集。它由六个Java开源项目的395个缺陷组成（2018年4月时，该数据集仍在更新当中）。每一个错误版本会恰好有一个缺陷。这个缺陷可能涉及多行甚至多个文件。与Siemens数据集相比，Defects4j数据集的缺陷和测试用例都更接近实际开发情况。 Savant[25]就是在Defects4j数据集上验证的。

第三章　问题分析

本章将基于频谱的缺陷定位和基于状态覆盖的缺陷定位应用在数据集Defects4j的某些缺陷上，分析这些技术在实际缺陷上的优点与不足。

3.1　使用基于频谱的缺陷定位

基于频谱的缺陷定位对代码的内容没有假设，所使用的信息只有语句的覆盖情况。考虑Defects4j中math项目的第五个缺陷，其代码如下：

```
1  public Complex reciprocal() {
2      if (isNaN) {
3          return NaN;
4      }
5
6      if (real == 0.0 && imaginary == 0.0) {
7          return NaN; // Faulty code
8                      // Should be "return INF;"
9      }
10     ...
11 }
```

为了使用基于频谱的缺陷定位，我们运行测试用例，并且收集语句的覆盖情况。针对第2，3，6，7行语句，得到语句的覆盖情况如表3.1所示。共有一个失败的测试用例。

可以发现，第3行肯定不是缺陷语句，因为它没有被失败的测试用例覆盖过。第2，6，7行都有可能是缺陷语句。这三行都是被一个失败测试用例覆盖。根据它们被覆盖的通过测试用例的个数，可以知道第7行最有可能出错，其次是第6行，最后是第2行。这是根据了基于频谱的缺陷定位的假设，即被失败的测试用例执行的语句，更有可能

语句	被覆盖的失败测试用例个数	被覆盖的通过测试用例个数
2	1	5
3	0	1
6	1	4
7	1	0

表 3.1　Defects4j中Math的第五个缺陷的测试用例覆盖语句的情况

a_{ef}	一个语句被失败的测试用例覆盖的次数
a_{nf}	一个语句未被失败的测试用例覆盖的次数
a_{ep}	一个语句被通过的测试用例覆盖的次数
a_{np}	一个语句未被通过的测试用例覆盖的次数
a_f	失败的测试用例的个数
a_p	通过的测试用例执行的次数

表 3.2　基于频谱的错误定位的数学符号及其意义

公式名称	公式
Ochiai[1]	$\text{Susp}(s) = \frac{a_{ef}}{\sqrt{a_f \times (a_{ef}+a_{ep})}}$
Tarantula[19]	$\text{Susp}(s) = \frac{\frac{a_{ep}}{a_p}}{\frac{a_{ep}}{a_p}+\frac{a_{ef}}{a_f}}$
Barinel[3]	$\text{Susp}(s) = 1 - \frac{a_{ep}}{a_{ep}+a_{ef}}$
DStar[44]	$\text{Susp}(s) = \frac{2}{a_{ep}+(a_f-a_{ef})}$
Op2[33]	$\text{Susp}(s) = a_{ef} - \frac{a_{ep}}{a_p+1}$

表 3.3　部分基于频谱的缺陷定位怀疑度公式

有错误。而被通过的测试用例执行的语句，更有可能是正确的。

为方便此后的表述，引入一些数学符号，见表3.2。表中的统计量就是程序频谱。

表3.3中列出了部分经典的怀疑度公式。这些公式都遵循被失败的测试用例执行的语句，更有可能有错误。而被通过的测试用例执行的语句，更有可能是正确的。

利用表3.3中的公式，计算第2，6，7行的怀疑度，得到表3.4。可以发现，这五个怀疑度公式都满足

$$\text{Susp}(7) > \text{Susp}(6) > \text{Susp}(2)$$

这五个怀疑度公式都认为第7行是最有可能出错的语句，而第7行也确实是出错的语句。这些公式之间的差距不同。比如Tarantula和Op2认为这三行的怀疑度是非常接近的。 Ochiai，Barinel和DStar认为第2行和第6行的怀疑度接近，而第7行的怀疑度明显高于第2行和第6行的怀疑度。

这个例子体现了基于频谱的缺陷定位准确定位错误的能力。但是实际上效果往往没有这么好。其实在该缺陷中，还存在一个正确的语句，它被一个失败的测试用例覆盖过，且从没有被正确的测试用例覆盖过。这个正确的语句的频谱信息和错误语句的频谱信息完全一致，所以它们的分数会相同。而这个正确的语句将会干扰开发者对缺陷的分析。

公式	第2行	第6行	第7行
Ochiai	0.4082	0.4472	1.0000
Tarantula	0.9988	0.9990	1.0
Barinel	0.1667	0.2000	1.0
DStar	0.2000	0.2500	Infinity
Op2	0.9988	0.9990	1.0

表 3.4　Defects4j中Math的第五个缺陷的经典公式怀疑度

3.2　使用基于状态覆盖的缺陷定位

考虑Defects4j数据集中Math的第二个缺陷，其代码如下：

```
1 public double getNumericalMean() {
2     return (double) (getSampleSize() * getNumberOfSuccesses()) / (
          double) getPopulationSize(); // Faulty code
3     // Should be "return getSampleSize() * (getNumberOfSuccesses() / (
          double) getPopulationSize())"
4 }
```

使用Ochiai方法的话，该错误语句被排到第11位。并列的分数将取其平均排名（期望排名），这是很多研究方法所使用的评估方式[36, 41, 46, 51]。事实上该语句的分数位列第2位，但是一共有17个语句和该语句分数并列，导致最终排名为第11位。基于频谱的缺陷定位方法在这里失效了，因为获取的信息不足。其实这个缺陷更加适合使用基于状态覆盖的缺陷定位技术。

状态覆盖就是使用谓词把具体状态划分为抽象状态。比如，对于如下代码，

```
1 a = Math.abs(a);
2 if (update_b) {
3     b = Math.sqrt(a);
4 }
```

当a和b的类型都为int时，如果a的值为最小的int时（a = -2147483648），则代码会在第3行出错（b的值为NAN）。这是因为当a = -2147483648时，第1行的a会被赋值为一个负数，于是在第3行进行sqrt操作的时候，就会出错。在第1行的时候，考虑两个抽象的状态a ≥ 0和a < 0。发现通过的测试只有a ≥ 0这个状态，而失败的测试只有a < 0这个状态。所以可以认为a < 0是缺陷状态，引入这个状态的第1行的语句很可能就是缺陷语句。通过谓词a ≥ 0和a < 0把程序的具体状态划分成了两个抽象状态，从而定位了第3行的缺陷。

t_f	一个谓词被观测为真的失败的测试用例的个数
t_p	一个谓词被观测为真的通过的测试用例的个数
a_f	一个谓词被观测的失败的测试用例的个数（谓词不一定为真）
a_p	一个谓词被观测的通过的测试用例的个数（谓词不一定为真）
F	失败的测试用例的个数
P	通过的测试用例执行的次数

表 3.5　统计性调试的数学符号及其意义

3.2.1　统计性调试

Liblit[26]提出的统计性调试使用预定义谓词。预定义谓词分为三类

- **分支**：对每一个条件语句，观察这个条件语句为真的谓词和为假的谓词。这个条件语句包括`if`条件这样的，也包括各种隐式的条件比如循环。
- **返回**：在C程序中，一个函数的返回值往往会被用于表达成功或者失败。对于每一个数值的返回值，观察六种谓词$< 0, \leq 0, > 0, \geq 0, = 0, \neq 0$。
- **数值对**：对于每一个数值赋值语句`x = ...`，找到所有和`x`同类型的、在作用域内的变量`y`和常量表达式`c`。对于每个`y`和`c`，观察六种谓词$<, \leq, >, \geq, =, \neq$。

为方便此后的表述，引入数学符号表3.5。

通过预定义谓词被测试用例的覆盖情况，计算得到每个谓词对应的怀疑度：

$$\text{StatisticalDebugging}(s) = \frac{2}{\frac{1}{\frac{t_f}{t_f+t_p} - \frac{a_f}{a_f+a_p}} + \frac{log(F)}{log(t_f)}}$$

当把统计性调试应用到Defects4j数据集中Math的第二个缺陷时，会在出错的代码处增加谓词，因为出错的代码处刚好是一个返回。虽然在Java程序中，函数返回值不会像C程序那样经常用于表达成功或失败，但是这些返回值有时也表达出程序执行的一些信息。在Math的第二个缺陷中，会发现该错误语句处的六个谓词会有表3.6中的覆盖情况。另外还有$a_f = 1$，$a_p = 6$。然而谓词1、2、5这些真分支被失败的测试用例覆盖的谓词的怀疑度都为0，谓词3、4、6的怀疑度都为负数。因为谓词1，2，5的$t_f = 1$，导致$log(t_f) = 0$，然后$\frac{log(F)}{log(t_f)} = INF$，于是最终计算得到的怀疑度为0。而谓词3、4、6由于$t_f = 0$，导致$log(t_f) = -INF$，致使分母中第二项为0。虽然谓词1、2、5的分数比3、4、6的分数高，但是0分并没有让这个出错的语句在整个代码的执行语句中排到前面。事实上对于所有真分支被失败用例覆盖的语句，由于其$t_f = 1$，最终其怀疑度都为0。由于每个谓词都存在和它取值相反的另一个谓词（比如`x > y`和`x <= y`），所以$t_f = 1$总是存在的。

在这个例子中，统计性调试得到了具有划分缺陷状态和非缺陷状态的谓词。但是

	谓词	谓词为真的失败的测试用例个数t_f	谓词为真的通过的测试用例个数t_p
1	`retValue < 0`	1	0
2	`retValue <= 0`	1	1
3	`retValue > 0`	0	5
4	`retValue >= 0`	0	6
5	`retValue != 0`	1	5
6	`retValue == 0`	0	1

表 3.6　返回值谓词的覆盖情况，其中
`retValue = (double) (getSampleSize() * getNumberOfSuccesses()) / (double) getPopulationSize()`

由于统计性调试需要多个失败的测试用例覆盖出错的语句，而在Defects4j这个数据集中多数缺陷都只有一个测试用例覆盖到，导致统计性调试的效果在Defects4j数据集上效果不好。在Liblit[26]的实验中，对每一个研究对象生成32000个随机输入。于是一个错误语句往往能被多个失败的测试用例覆盖。可见统计性调试的方法在实际数据集里往往只有一个失败的测试用例的情况下并不适用。

3.2.2　SOBER

SOBER[28]也是基于状态覆盖的错误定位，它改进了统计性调试的怀疑度计算方法。SOBER的公式计算的是对一个谓词，在失败的测试用例下这个谓词为真的概率分布，和在通过的测试用例下这个谓词为真的概率分布是否相似。如果概率分布无论是在失败的测试用例中还是通过的测试用例中都一样，那么这个谓词对应的变量等和缺陷的关系就越小。如果两个概率分布相差很大，说明这个谓词对应的抽象状态很有可能就有缺陷状态。引入这个缺陷状态的语句很可能就是出错的语句。

SOBER的计算公式为

$$\text{Sober}(s) = -log(\text{Sim}(f(X|\theta_p), f(X|\theta_f)))$$

其中$f(X|\theta_p)$表示通过的测试用例下这个谓词为真的概率分布，$f(X|\theta_f)$表示失败的测试用例下这个谓词为真的概率分布，Sim函数则计算这两个概率分布的相似度。

为了计算相似度，首先提出零假设：

$$\mathcal{H}_0 : f(X|\theta_p) = f(X|\theta_f)$$

即两个概率分布没有区别。然后使用总平均μ和方差σ^2来刻画概率分布，所以零假设为 $\mu_p = \mu_f$并且$\sigma_p^2 = \sigma_f^2$。假设一共有m个失败的测试用例，令$\mathbf{X} = (X_1, X_2, ..., X_m)$是一

测试用例编号	覆盖真分支的次数	覆盖假分支的次数	当前测试用例状态
1	0	2	通过
2	0	2	通过
3	0	2	通过
4	0	2	通过
5	1	0	失败
6	0	10	通过
7	0	1000	通过

表 3.7　SOBER方法下，Defects4j的Math第二个缺陷的错误语句里，分数最高的谓词的覆盖情况

个从$f(X|\theta_f)$得到的独立同分布随机样本。在零假设下，根据中心极限定理，统计量

$$Y = \frac{\sum_{i=1}^{m} X_i}{m}$$

渐近于$N(\mu_p, \frac{\sigma_p^2}{m})$，一个均值为$\mu_p$方差为$\frac{\sigma_p^2}{m}$的正态分布。令$f(Y|\theta_p)$为$N(\mu_p, \frac{\sigma_p^2}{m})$的概率密度函数。使用似然函数$L(\theta_p|Y)$作为相似度计算的函数，有

$$\text{Sim}(f(X|\theta_p), f(X|\theta_f)) = L(\theta_p|Y) = f(Y|\theta_p)$$

根据正态分布的性质，统计量

$$Z = \frac{Y - \mu_p}{\sigma_p/\sqrt{m}}$$

渐近于$N(0,1)$，而且

$$f(Y|\theta_p) = \frac{\sqrt{m}}{\sigma_p}\varphi(Z)$$

其中$\varphi(Z)$是$N(0,1)$的概率密度函数。最后得到怀疑度计算公式：

$$\text{Sober}(s) = log\left(\frac{\sigma_p}{\sqrt{m}\varphi(Z)}\right)$$

从SOBER的公式可以看出，SOBER仍然是建立在有大量测试用例的基础上。少量的测试用例会让概率分布不能准确反映出谓词真假分支的取值分布。SOBER的验证实验也是在人造的Siemens数据集上完成的。

在Defects4j的Math的第二个缺陷这个例子中，该错误语句的六个谓词中，`((double) (getSampleSize() * getNumberOfSuccesses()) / (double) getPopulationSize()) < 0`得分最高，覆盖情况见表3.7，怀疑度为360.85。在怀疑度列表中排名第10，效果并不理想。

3.3　分析结论

在以上的例子和分析中我们发现，现有缺陷定位存在不足。

对于基于频谱的缺陷定位来说，它仅仅依赖频谱信息去区分正确语句和错误语句，会导致很多正确语句也具有很高的怀疑度。特别地，如果一个正确语句只被失败的测试用例覆盖，那么它将拥有非常高的怀疑度。这是由于频谱信息的信息量太少，基于频谱的缺陷定位忽略了程序状态等被基于状态覆盖的缺陷定位关注的信息。

而基于状态覆盖的缺陷定位，虽然能够获得比频谱信息更多的信息，但是现有的方法都依赖于大量的测试用例。在测试用例不足的时候，基于状态覆盖的缺陷定位无法给出具有区分度的怀疑度。

第四章　设计

4.1　结合基于频谱的缺陷定位和基于状态覆盖的缺陷定位

既然基于频谱的缺陷定位和基于状态覆盖的缺陷定位各有优劣，那么是否可以结合这两种缺陷定位的方法呢？事实上已经有研究[25, 51]，结合了多种缺陷定位方法，并且获得了比较好的结果。但是这些研究的结合方式都是在比较高的层次，比如使用机器学习方法对不同缺陷定位得到的结果进行组合。这样的结合方式会有两个缺点。一是他们没有解释为什么他们的方法会起作用。二是他们没有深入理解缺陷定位方法起作用的原因，仅仅是把各个方法的结果合在一起。

所以，本文试图提出一个能够结合现有缺陷定位（比如基于频谱的缺陷定位和基于状态覆盖的缺陷定位）的方法去改进缺陷定位技术，同时本文试图解释这个结合为什么起作用的原因。

虽然在直觉上我们认为基于频谱的缺陷定位和基于状态覆盖的缺陷定位是完全不一样的。因为基于频谱的缺陷定位依靠的是程序元素的覆盖情况，而基于状态覆盖的缺陷定位依靠的是用谓词来划分状态。但是事实上这两种缺陷定位技术有相似的地方，它们都是通过一个公式对程序元素打分。基于频谱的缺陷定位的频谱信息，其实相当于是对每一个语句都关联了一个`true`这样的谓词。这样看来基于频谱的缺陷定位相当于基于状态覆盖的缺陷定位的一个特殊情况。而基于状态覆盖的缺陷定位收集的谓词的覆盖信息也可以看做是程序频谱信息的一种，所以基于状态覆盖的缺陷定位也可以看做基于频谱的缺陷定位的一个特殊情况。

考虑3.2章中基于状态覆盖的缺陷定位的例子。统计性调试和SOBER都无法给出很好的定位结果。但是当观察统计性调试的覆盖情况3.6，我们却可以“猜测”出当前语句很可能是错误语句。这是因为我们带入了基于频谱的缺陷定位的假设：被失败的测试用例执行的语句，更有可能有错误。而被通过的测试用例执行的语句，更有可能是正确的。根据这个假设，表3.6中的谓词3、4、6都不太可能是能够划分出缺陷状态的谓词，因为它们都没有被失败的测试用例覆盖过。谓词1最有可能是能够划分出缺陷状态的谓词，其次是谓词2，最后是谓词5。这是因为谓词1、2、5都被一个失败的测试用例覆盖过，而谓词1没有被通过的测试用例覆盖过。这种情况下被越少的通过的测试用例覆盖，越有可能就是能够划分出缺陷状态的谓词。怎样去具体地表示这个怀疑度呢？这其实是基于频谱的缺陷定位解决的问题了，那就是使用怀疑度公式。使用Ochiai怀疑度公式去计算表3.6中谓词的怀疑度，得到表4.1。可见谓词1以1.0000的分

数远远高于其他谓词，成为怀疑度很大的谓词。对于每个语句的多个谓词，采用怀疑度最高的谓词的分数作为这个语句的怀疑度。使用Ochiai怀疑度公式，计算Math的第二个缺陷的各个谓词怀疑度，缺陷语句排名第3（第1到4名并列），相比于基于频谱的状态覆盖第11位、统计性调试全部为0和SOBER第10的结果，有显著提升。

不仅如此，这样得到的结合了基于频谱的缺陷定位和基于状态覆盖的缺陷定位的结果，还可以和原始的基于频谱的缺陷定位的结果相结合。比如Math的第二个缺陷的错误语句，其基于频谱的缺陷定位的怀疑度为0.3780，加上结合后的谓词怀疑度1.0000，总怀疑度为1.3780，在怀疑度列表中排名第2。效果好于原始的定位结果和结合后的结果。

基于频谱的缺陷定位和基于状态覆盖的缺陷定位的方法都可以表述为，给定一个程序元素e，一个产生谓词的函数s，和一个怀疑度计算公式r，对于这个程序元素e，它的分数函数c可以表述为：

$$c(s, e, r) = max_{p \in s(e)} r(p)$$

对于这个程序元素e，首先通过$s(e)$获取其所有谓词。对于基于频谱的缺陷定位方法这个谓词恒为`true`。然后对每个谓词p计算其在怀疑度公式r下的怀疑度$r(p)$。这个怀疑度公式包括基于频谱的缺陷定位和基于状态覆盖的缺陷定位怀疑度公式。上述例子提出的结合的方法，使用的基于频谱的缺陷定位的r，和基于状态覆盖的缺陷定位的s。这样可能会得到多个怀疑度，采用某种方式把它们结合起来（这里使用的最基本的max函数作为结合怀疑度的方法），就可以得到这个程序元素的最终分数。

本文提出两种结合方法进行实验：

- **MaxPred**

  对于一个程序元素e，在某个怀疑度公式r和谓词$s(e)$下，计算得到的多个怀疑度分数，并取其最大值作为e的怀疑度。

- **LinPred**

  给定两组谓词$s_0(e)$和$s_1(e)$，使用MaxPred计算得到 e的两个怀疑度c_0和c_1。再利用一个线性模型结合两个怀疑度，得到$C(e) = (1 - \alpha) \times c_0 + \alpha \times c_1$。

4.2 预定义谓词和预测谓词

在统计性调试和SOBER中，都使用的是预定义的谓词。一个谓词的好坏决定了能否划分出缺陷状态。

考虑Defects4j中Math的第四个缺陷，其代码如下：

	谓词	Ochiai分数
1	`retValue < 0`	1.0000
2	`retValue <= 0`	0.7071
3	`retValue > 0`	0.0000
4	`retValue >= 0`	0.0000
5	`retValue != 0`	0.4082
6	`retValue == 0`	0.0000

表 4.1　使用Ochiai计算谓词怀疑度，其中
`retValue = (double) (getSampleSize() * getNumberOfSuccesses()) / (double) getPopulationSize()`

```
1  public Vector3D intersection(final SubLine subLine, final boolean
       includeEndPoints) {
2      // compute the intersection on infinite line
3      Vector3D v1D = line.intersection(subLine.line);
4  +   if (v1D == null) {
5  +       return null;
6  +   }
7
8      // check location of point with respect to first sub-line
9      Location loc1 = remainingRegion.checkPoint(line.toSubSpace(v1D));
10     ...
11 }
```

第4，5，6行是修复缺陷的代码。这个缺陷是缺少了对变量`v1D`是否是空的判断。考虑谓词`v1D == null`，会发现通过的测试用例都不会覆盖这个谓词，只有失败的测试用例覆盖这个谓词。因为一旦这个谓词为真，那么后续某些使用这个`v1D`变量的操作就会造成空指针的错误。

然而这个谓词并不能由预定义的谓词得到。但是事实上代码中其他的地方存在`var == null`这样的判断。于是本文提出了一种基于机器学习的预测谓词的方法，来更准确地找出划分缺陷状态的谓词。

或许我们不使用机器学习方法，而是把`var == null`这样经典的判断加入预定义谓词呢？但问题是这样的谓词永远是加不完的。而且对于每个程序，它们可能还拥有跟自己上下文有关的独特的谓词。所以从它们自己的代码中来学习出谓词是更有效的方法。

4.3 缺陷定位框架

本文提出了结合了基于频谱的缺陷定位和基于状态覆盖的缺陷定位的缺陷定位方法，同时还提出了一种机器学习方法去预测谓词。

本文的缺陷定位框架主要包括以下几步

- **收集特征** 当使用预定义谓词时不需要这一步。假如缺陷代码是版本v，则从版本v的源代码中提取特征。特征分为两种，一种是变量预测模型的特征，一种是谓词预测模型的特征。
- **训练模型** 当使用预定义谓词时不需要这一步。把得到的特征放入机器学习模型中训练。两种特征分别单独训练，得到一个预测变量出现在谓词中概率的模型（简称VAR模型），和一个根据一个变量预测可能出现哪些谓词的模型（简称EXPR模型）。
- **收集失败测试用例覆盖的语句** 只有被失败的测试用例覆盖的语句才有可能是错误的语句。因为只需要对失败测试用例覆盖的语句收集其谓词的覆盖情况就可以了。
- **获取谓词** 如果使用预定义谓词，则根据预定义谓词的规则分析代码，得到谓词。如果使用预测谓词模型，则提取需要插入谓词的相关变量和语句的特征，放入已经训练好的模型中，预测出当前位置最有可能出现的N个谓词。
- **收集谓词覆盖情况** 把谓词插入代码中，并执行测试用例，收集谓词的覆盖情况。不同怀疑度公式收集的覆盖情况可能会有不同。覆盖情况包括：谓词的真（或假）分支被多少个失败（或通过）的测试用例覆盖，谓词（无论是真分支还是假分支）被多少个失败（或通过）的测试用例覆盖，谓词的真（或假）分支在一个失败（或通过）的测试用例中被覆盖的次数等等。
- **计算怀疑度** 根据上一步收集的谓词覆盖情况，带入基于频谱的缺陷定位和基于状态覆盖的缺陷定位的怀疑度公式计算怀疑度。收集语句的覆盖情况，带入怀疑度公式计算怀疑度，并且与谓词怀疑度结合。

4.4 基于机器学习的预测谓词模型

对谓词的预测分为三步：

- 第一步，使用VAR模型预测语句中的某个变量出现在谓词中的概率P_{var}。
- 第二步，使用EXPR模型预测语句中的某个变量会出现在谓词 $pred_i$ 中的概率 P_{pred_i} 。
- 第三步，将谓词按照$P_{var} \times P_{pred_i}$排序。

特征名称	特征说明
FileName	文件名
MethodName	函数名
VarName	变量名
VarType	变量类型
LastAssign	变量最后一次赋值的操作
Dis0	变量声明和使用之间的距离
PreAssNum	变量此前赋值的次数
IsParam	变量是否是方法的参数
InFor	变量是否出现在循环中
InCondNum	变量在条件中出现的次数
BodyUse	变量在条件语句的语句体里的使用情况
OutUse	变量在条件外的使用方式

表 4.2　VAR模型和EXPR模型特征

第一步中的变量是赋值表达式的左值。也就是说我们只会对失败的测试用例覆盖的赋值语句进行表达式的预测。

4.4.1　机器学习特征

对于一个有缺陷的代码，我们从当前这个版本的所有源代码中提取特征。

使用的变量特征和谓词特征如表4.2。不同的是VAR模型的标签是这个变量是否出现在条件中， EXPR模型的标签是这个变量相关的谓词是什么。也就是说VAR模型使用的样本来自于所有变量，而EXPR模型使用的样本仅来自于出现在谓词中的变量。

这些特征有的是认为这些特征相似度更高的变量更可能会有相似的属性。相似的文件名、函数名可能是实现着相似的功能，比如可能是继承了同一个父类的子类。变量名相似也可能是有相似的功能，比如 `len` 和 `length`。类型相同同理，比如对一个 `Object` 类型的变量进行是否为空指针的判断。相似的功能可能就会使用相似的谓词。还有特征是描述这个变量的上下文信息，这些特征也会影响谓词。比如 LastAssign 说明了这个变量值是如何得到的，比如从`getMin()`函数得到的值v容易出现在谓词中，并且谓词很可能是`v < 0`这样的比较。有的特征直接描述这个变量在条件表达式中的情况，比如 InFor， InCondNum等。

4.4.2　机器学习特征编码

编码

对于数值型的变量，直接使用其值作为特征。而对于分类变量（categorical variable），虽然其值可能表现为0、1、2…… 这样的数字，但是其实并不存在0 < 1 < 2这

样的大小关系。直接使用其值会让模型以为这个变量存在大小关系。所以对于分类变量，本文采取独热（one-hot）编码。比如对于一个值为0、1、2的分类变量v，使用新的变量v_0、v_1、v_2代替v。其中$v_i = 1$表示$v = i$，而$v_i = 0$表示$v \neq i$。

字符串特征编码

对于字符串类型的特征，比如文件名、函数名和变量名，本文也会使用独热编码。但是直接使用独热编码会有两个问题：

1. 特征维度过大。比如对Defects4j中Math项目的第一个缺陷来说，仅不同的方法就有795个。这意味着如果把方法改成独热编码，将会把一个1维的特征变成795维的特征。这样可能造成维度灾难[7]。虽然一开始随着特征数的上升，机器学习模型的预测效果也会上升，但是当维度过高的时候，实际性能是下降的。但这还不是最关键的因素。
2. 丢失了字符串内部的特征。字符串的变量和其它的变量不同，它们之间还有内部的特征。比如变量名len和length之间的相似度比len和domain之间的相似度要高，文件名SubLine.java和Line.java之间的相似度比SubLine.java和BracketFinder.java之间的相似度高。简单地把不同字符串看成完全独立地不同特征会丢失很多有用的信息。

对字符串的编码采取三步。

首先是将字符串转换为向量。对变量名，采取一种类似2-gram的方法。每一个变量名都会被转成一个长度为729(27×27)的一维向量。变量名中的大写字母会先被转为小写字符。对转换后的变量名$s_1 s_2 ... s_n$，考虑每两个相邻字符的字符串$s_1 s_2$，$s_2 s_3$ $\cdots\cdots$ $s_{n-1} s_n$。这个向量的第i位为1表示变量名中存在两个相邻字符$s_j s_{j+1}$，满足$f(s_j) \times 27 + f(s_{j+1}) = i$。 f是一个映射函数，将一个字符转换成一个0到26之间的数字。对于f有$f('a') = 0, f('b') = 1, ..., f('z') = 25$，然后a到z以外的字符都被转为26。这样的话len和length的特征向量之间就会有两位相同，而len和domain之间则完全没有相同的。对函数名和文件名，去掉后缀，然后利用Java中使用的驼峰命名法将它们按照驼峰分割开来。比如 SubLine.java 会被拆为 Sub 和 Line，而 Line.java 得到 Line，BracketFinder.java 得到Bracket和Finder。收集所有分割后的词（函数名和文件名分开收集），假设有N_{func}和N_{file}个，则每个函数名（或文件名）就转成一个长度为N_{func}（或N_{file}）的向量。这个向量的第i位为1表示函数名（或文件名）中含有字符串$g_{func}(i)$（$g_{file}(i)$）。 g_{func}和g_{file}是向量下标和字符串的映射。比如 $g_{file}(57) = "Line"$ 的话，Line.java的向量的第57位为1，其他位为0。

然后是对字符串转换后的向量进行聚类。聚类使用的是K-means聚类算法。因为

不同的项目所涉及的变量名、函数名、文件名数量差别较大，不适合使用单一的某个常数作为聚类的类别数。所以聚类的类别数为$Unique(names)/c$。其中$names$表示所有变量名或函数名或文件名，$Unique(x)$表示x中不重复的值的数量，c是某个常数。

最后是根据聚类给字符串编码。假如聚类结果有N个类，那么就把这N个类编号为1到N。每个类中的每个字符串的编号等于它所处的类的编号。每个字符串的编号就是它的特征值，对这个特征值使用独热编码就得到最终的特征。于是通过重新编码和聚类，字符串特征的内部特征被抽取出来，并且字符串特征的维度也被大幅减小。

预测时的特征新值

在预测的时候，特征里面可能会有在训练的特征里面没有出现过的值。如果这个值是数值型的值，那么直接使用这个值就可以。但是如果是经过编码之后的值，就需要给这个新值一个编码。

为了给出一个和训练时编码一致的编码，需要保存训练时的部分数据。包括变量名、文件名、函数名的聚类模型，文件名、函数名中向量下标和字符串的对应函数g_{func}、g_{file}，训练时使用的独热编码。

对于新的变量名，按照训练时的处理方法将其转为729(27×27)的一维向量。然后计算出这个向量与训练时变量名聚类模型中的哪个中心点距离更短，把这个变量名划入这个中心点的分类。最后使用训练时变量名的独热编码给这个分类编码即可。

对于新的文件名或函数名，有两种情况。第一种情况是，这个文件名虽然没有出现过，但是它按照驼峰拆分之后的字符串都出现过。这种情况下使用g_{func}和g_{file}直接构造特征向量。第二种情况是，这个文件名按照驼峰拆分之后的字符串有没有出现过的。没有出现过的字符串则会被忽略。构造出的特征向量使用训练时的聚类模型划分分类，最后使用训练时的独热编码给改分类编码。

对于除变量名、文件名、函数名以外的分类变量，如果出现了新的值，假设这个分类变量的类别有C个，那么新值转成独热编码后其特征向量为C个0。

4.4.3　机器学习模型

本文在小量数据上尝试了多种机器学习模型，最后选定了两种机器学习模型：全连接神经网络和决策树模型。

神经网络模型

VAR模型和EXPR模型使用同样的全连接神经网络。这个神经网络由一个输入层，六个隐藏层，一个输出层，一个softmax层构成。输入层的神经元数量和特征数量一

致，输出层的神经元数量和分类数量一致。对于VAR模型来说输出层的神经元有两个，对于EXPR模型来说输出层的神经元数量和可能的谓词数量一样。六个隐藏层每层都是64个节点。这个是小部分实验之后得出的比较适合的值。采用的激励函数是广泛使用的线性整流函数（ReLU）。

$$\mathrm{ReLU}(features) = max(features, 0)$$

神经网络使用误差逆传播算法进行训练，优化算法使用 Proximal Adagrad 算法。 Proximal Adagrad 算法能够在训练中自动对学习率进行调整，并且能够通过正则化防止过拟合。损失计算采用的softmax cross entropy。

假如分类数是c，softmax层的输入是一个$c \times 1$的向量，输出也是一个$c \times 1$的向量，表示当前输入的标签是各个类别的概率。记原始的输入向量是$(a_1, a_2, ..., a_c)$，softmax层的输出是 $(S_1, S_2, ..., S_c)$，则有

$$S_j = \frac{e^{a_j}}{\sum_{k=1}^{c} e^{a_k}}$$

以softmax层的输出计算softmax cross enptropy：

$$E = -\sum_{j=1}^{c} y_j log(S_j)$$

其中y_j是输入对应的真实的标签。

为避免过拟合，本文采用了早停（early stop）的方法。训练时会把训练数据分为训练集和验证集。使用训练集进行训练，然后不断评估当前模型的训练集的损失和验证集的损失。当训练集的损失不断下降，但是验证集的损失却开始上升时，说明模型出现了过拟合，停止训练。然后把训练过程中验证集损失最小时的模型作为最终模型。

决策树模型

决策树是一种常用的机器学习方法。顾名思义，决策树是一棵树，包含一个根节点、若干个内部节点和叶节点。每个叶节点对应一个决策结果，内部节点和根节点则对应于一个属性测试。根据节点的属性测试，样本被划分到对应的子节点中。其学习流程的基本思想是分治法。

决策树中需要依靠节点的纯度来选择最优划分属性。本文使用的基尼指数来选择划分属性。假设当前样本集合为D，其中第k类样本所占比例为$p_k, (k = 1, 2, ..., |\gamma|)$，则

基尼指数为：

$$\text{Gini}(D) = \sum_{k=1}^{|\gamma|} \sum_{k' \neq k} p_k p_{k'} = 1 - \sum_{k=1}^{|\gamma|} p_k^2$$

基尼指数反映了从数据集中随机选取两个样本，其类别标记不一致的概率。所以基尼值越小，纯度越高。所以选择使得划分后基尼指数最小的属性作为最优划分属性。

树的深度没有限制，也就是说节点会被一直展开直到所有叶节点都是纯净的节点（即所包含的样本都只属于同一个分类）或叶节点的样本数量小于2。

4.5　收集频谱信息

代码插装是一种常用的记录程序执行情况、修改程序的方法。采用的工具是eclipse提供的Java Development Tool①，简称JDT。JDT不仅可以构造给定Java代码的抽象语法树，还可以新建、修改、插入、删除抽象语法树从而新建、修改、插入、删除Java代码。本章中很多实现都依赖于JDT。

本文使用代码插装技术主要目的是收集频谱信息。比如为了收集失败测试用例覆盖的语句，可以在每个语句后面加上一个打印语句。这个打印语句会打印出当前的文件名和对应语句的行数。用失败的测试用例执行这个插装后的程序，就可以得到失败测试用例覆盖的语句。

对于谓词P和一个测试用例T_i，可能需要收集四种信息：

1. P是否被T_i观察过（无论是真还是假）。
2. T_i中P是否至少一次为真。
3. T_i中P为真的次数。
4. T_i中P为假的次数。

对于基于频谱的缺陷定位的公式来说，需要第2个信息。对于统计性调试的公式来说，需要第1、2个信息。对于SOBER的公式来说，需要第3、4个信息。所以整体来说谓词的插装有两种形式。第一种用于基于频谱的缺陷定位公式和统计性调试公式：

```
1 // predicateSignature is a string that uniquely represents the
     predicate.
2 SpecLogger.observe(predicateSignature);
3 if (predicate) {
4     SpecLogger.cover(predicateSignature);
5 }
```

① http://www.eclipse.org/jdt/

SpecLogger.observe记录这个谓词为观察过（信息1），SpecLogger.cover记录这个谓词为真过（信息2）。第二种用于SOBER公式：

```
1 // predicateSignature is a string that uniquely represents the
     predicate.
2 if (predicate) {
3     SpecLogger.coverTrueBranch(predicateSignature);
4 } else {
5     SpecLogger.coverFalseBranch(predicateSignature);
6 }
```

SpecLogger.coverTrueBranch记录这个谓词为真的次数（信息3），SpecLogger.coverFalseBranch记录这个谓词为假的次数（信息4）。

这几个SpecLogger的函数可以是把自己内部的某个布尔型的成员变量置为真或假，也可以是把自己内部的某个整型的成员变量加一。这个SpecLogger对象在每个测试用例开始的时候重置（使用静态方法变量），并在测试用例结束的时候以某种格式将自己记录的信息输出到文件。

```
1 private void testSomething() {
2     SpecLogger.reset();
3     SpecLogger.testStatus = true;
4
5     // test ...
6     ...
7
8     SpecLogger.dump();
9 }
```

第三行是根据当前测试用例testSomething是通过的测试用例还是失败的测试用例而插装的。最后SpecLogger.dump把记录的信息输出到文件。其他程序从这个输出文件中可以构造出频谱信息。

4.6 不改变程序状态地插入谓词

统计性调试中自定义的谓词和预测出来的谓词可能有副作用。使用上一章中的插装方法会改变程序的执行状态，所以需要不改变程序状态地插入谓词。

统计性调试中的谓词要么是两个变量构成的二元表达式，要么就是本来就会在程序中执行的条件表达式。前者不会有副作用，而后者即使有副作用，由于其本来就要

在原程序中执行一次，因此只要利用执行的这一次的结果就可以。

预测出来的谓词可能含有有副作用的函数，或者含有赋值语句。于是对预测出谓词的静态分析，只留下一定无副作用的谓词。比如除了部分指定函数（如size），含有其他函数的谓词都会被过滤掉。

插入谓词仍然使用JDT。

4.6.1　插入预测的谓词

通过机器学习模型，每个语句可能会关联一组谓词。这些谓词会被机器学习模型赋予出现的概率。最终每个语句我们选择概率最高的5个谓词作为需要插入的谓词。

在插入预测的谓词之前，要先对谓词进行过滤。过滤包括两步：

1. 静态分析过滤掉可能不合法的谓词（比如对一个`int`类型的变量进行下标访问）。
2. 静态分析过滤掉可能有副作用的谓词。
3. 过滤掉不能编译的谓词。

一个预测出来的谓词分为两部分，一个是谓词$P(x)$，一个是变量v，最终构成谓词$P(v)$。一个谓词会被判定为不合法如果它满足以下至少一点：

- 含有数组访问`v[i]`且`v`并不是数组类型。
- 含有变量访问`v.a`且`v`是一个基本数据类型（比如`int`）的变量。
- 含有变量访问`a.v`且`v`不是`a`的一个域。
- {TODO:simple name}
- 含有函数调用`v.a()`且`v`是一个基本数据类型变量。
- 含有函数调用`f()`且`f`不属于预定义的合法函数（如`size`，`length`，`toString`，`contains`，`containsKey`，`Math.abs`，`Double.isInfinite`，`Double.isNaN`）。
- 含有域访问`v.a`且`v`是一个基本数据类型的变量。
- 含有中缀表达式`a op b`且`a`,`b`的类型和运算符`op`不匹配（比如对非数字类型进行加法）。

过滤有副作用的谓词主要是过滤掉以下几种：

- 含有前缀表达式，且其中运算符为`++`或`--`。
- 含有后缀表达式。
- 含有赋值语句。

最后单独地插入每一条谓词，过滤掉不能顺利编译的。在没有判定谓词是否合法的时候，也可以通过编译来排除不合法的谓词。但是由于谓词数量较多，通过预处理去掉部分肯定不对的谓词可以加速整个流程。

最后对于一个语句s，我们可以得到一组合法无副作用的谓词$P_1, P_2 \ldots$。我们再加

入取反的谓词$!P_1, !P_2, \ldots$。

然后就是将收集谓词频谱信息的代码插入到语句前或后。语句前后都插入的有`while`，`for`，`do`，插入在语句后面的有赋值语句，插入在语句前的有`if`，`switch`等其他所有语句。

4.6.2 插入预定义的谓词

预定义的谓词也可能有副作用，比如赋值语句`a[b++] = c`的谓词`a[b++] > d`，如果使用此前的插装方法，则插入`if (a[b++] > d)`这样的语句会产生副作用。但是如果使用中间变量，则上述代码可以重写为：

```
temp = c;
a[b++] = temp;
if (temp > d) {
    ...
}
```

所以预定义谓词的三种情况，分支、返回、数值对，都可以使用中间变量这样的方式来插入谓词，从而避免了副作用的情况。

由于分支对应多种情况，使用中间变量会比较复杂。分支的谓词是分支中含有的条件表达式。`DoStatement`中的条件表达式的值每次循环都会更新，再考虑上`continue`这样的语句，会让条件表达式的更新逻辑非常复杂。为了简单地插入谓词，并且由于条件表达式的类型都是布尔型，使用一个函数`logConditionCoverage`替换条件表达式。原代码为：

```
// example.java:
while(!iter.isEmpty()) {
        ...
}
```

插装了收集谓词频谱信息的语句后，代码被修改为：

```
// example.java
while(SpecLogger.logConditionCoverage(!iter.isEmpty(), "!iter.isEmpty()
   ", "!(!iter.isEmpty())") {
        ...
}

// SpecLogger.java
```

```
7  public static boolean logConditionCoverage(boolean condition, String
       trueLogInfo, String falseLogInfo) {
8          observe(trueLogInfo);
9          observe(falseLogInfo);
10         if (condition) {
11                 cover(trueLogInfo);
12         } else {
13                 cover(falseLogInfo);
14         }
15         return condition;
16 }
```

这里同时记录了假分支的覆盖情况。这样做有两个目的：

- 统计性调试中预定义的分支型谓词有条件表达式取反。
- 和预测谓词加入了谓词取反的情况保持一致。

4.7　计算怀疑度

收集频谱信息后，就可以带入计算怀疑度。无论是机器学习得到的谓词，还是预定义的谓词，都使用表3.3中的五种公式，以及统计性调试和SOBER总共七种公式进行计算。通过谓词计算得到的语句s_i的怀疑度记为s_{i1}。

除了能够使用谓词计算得到怀疑度以外，利用原始的基于频谱的缺陷定位方法和基于状态覆盖的缺陷定位方法（SOBER除外），还可以计算得到语句的怀疑度s_{i0}。本文尝试结合这两种怀疑度，比如使用$s_i = (1-\alpha) \times s_{i0} + \alpha \times s_{i1}$作为最终怀疑度。

第五章　实现

本章主要介绍{TODO:。。}

5.1　整体架构

章4.3中已经描述了本文要实现的缺陷定位框架。这个缺陷定位框架如图5.1所示，一共包含三个大的模块：谓词生成模块，频谱信息收集模块，和怀疑度计算模块。

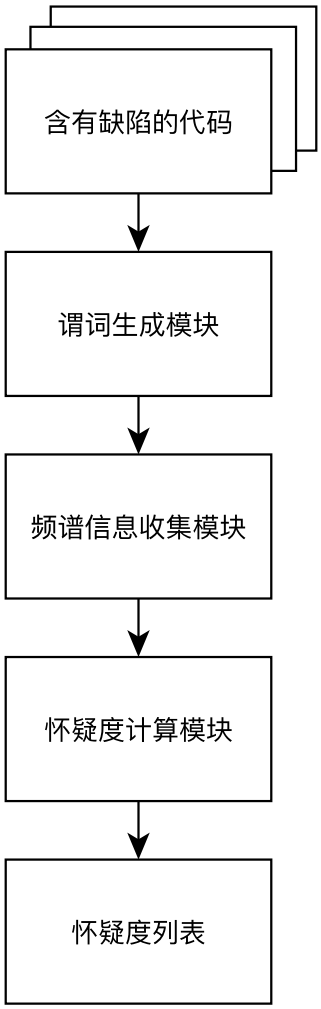

图 5.1　缺陷定位框架模块图

谓词生成模块生成预定义的谓词，或者利用机器学习模型预测出谓词。频谱信息收集模块把生成的谓词插装到缺陷代码，执行测试用例，收集测试用例覆盖谓词真假

分支的情况。最后怀疑度计算模块利用频谱信息，带入公式中计算怀疑度。

具体的流程图如图5.2所示。下面将会具体讲述这些步骤的实现。

5.2 谓词生成模块

谓词生成模块负责生成后续会使用的谓词. 谓词生成模块分为两种，一种是生成预测谓词，一种是生成预定义谓词。生成谓词的语句都是被失败的测试用例覆盖的语句。

5.2.1 生成预测谓词模块

生成预测谓词使用的是机器学习模型，使用Java和Python实现。首先，Java代码通过JDT遍历缺陷代码的抽象语法树，提取表4.2中的特征。这些特征被写入到磁盘上的 csv 文件中。 Python程序则读入 csv 文件中的特征，对特征进行编码和训练。

编码阶段，先对变量名、文件名、函数名使用scikit-learn[①]的K-Means聚类，其值转为类编号。聚类模型等被存储到磁盘。使用 scikit-learn 的 `LabelEncoder` 对特征 FileName，MethodName，VarName，VarType，LastAssign，BodyUse 和 OutUse 进行编码，归一化为数字，然后使用 `OneHotEncoder` 转为独热编码。

训练阶段，特征被放入神经网络或者决策树中。神经网络使用的全连接神经网络，用TensorFlow[②]的 `DNNClassifier` 实现。 TensorFlow 是一个采用数据流图用于数值计算的开源软件库。图中的节点表示数学操作，图中的线则表示在节点间相互联系的多维数据数组。网络的输入使用 TensorFlow 的 `numpy_input_fn`。使用这个函数的好处是，输入不需要在一开始的时候就被存储在内存，而是在需要的时候才生成一个批（batch）的数据。每次训练以一个批的数据为一次迭代。 VAR模型和EXPR模型的批尺寸（batch size）都为128。输入数据会被随机打乱（shuffle）。一个周期（epoch）就是把全部数据输入神经网络训练一次。输入数据按照7 : 3的比例分为训练集和验证集。训练时，会先使用训练集训练`INIT_EPOCHS`个周期。这是因为训练开始时，损失变化较大。此后每次都再输入一个周期，并收集训练集和验证集在输入这个周期后的新模型上的损失和准确率。程序会收集最近的`1`到`2 * TEST_EPOCHS`个周期的训练集和验证集的损失，记为$L(i,j)$。其中i表示是最近的第i个周期的损失，j为$true$的话表示是训练集的损失，为$false$表示是验证集的损失。然后分别计算`1`到`TEST_EPOCHS`，`TEST_EPOCHS`

① http://scikit-learn.org/

② https://www.tensorflow.org/

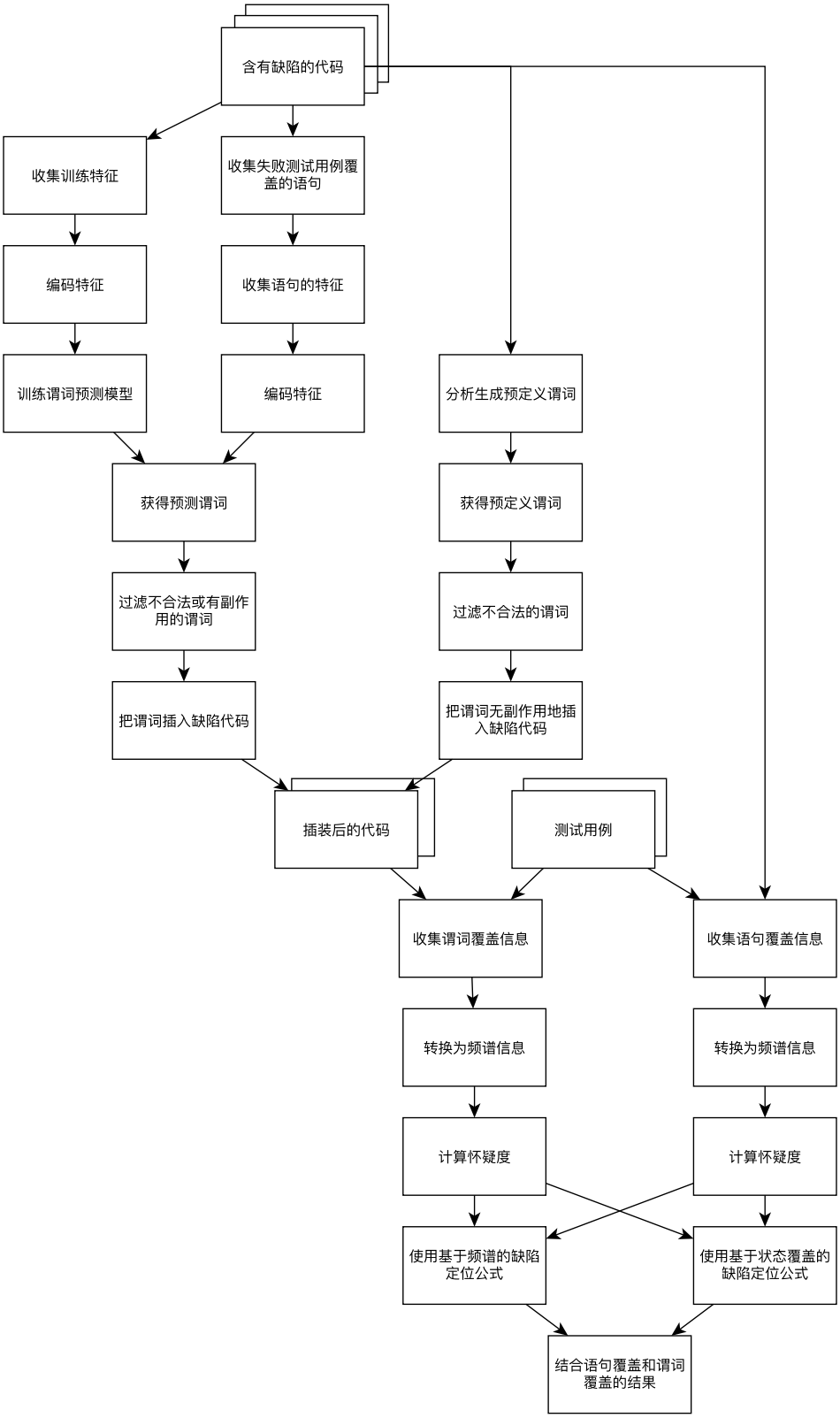

图 5.2　缺陷定位整体流程图

+ 1到2 * TEST_EPOCHS 的训练集总损失值

$$Loss_1 = \text{Loss}(1, TE, true)$$

$$Loss_2 = \text{Loss}(TE + 1, 2 * TE, true)$$

和验证集的总损失值

$$Loss_3 = \text{Loss}(1, TE, false)$$

$$Loss_4 = \text{Loss}(TE + 1, 2 * TE, false)$$

。其中TE表示TEST_EPOCHS， Loss定义如下：

$$\text{Loss}(start, end, j) = \sum_{i=start}^{end} L(i, j)$$

。终止训练的条件为

$$Loss_1 < Loss_2 \text{ and } Loss_3 > Loss_4$$

。也就是说当训练集的损失仍在下降，但是验证集的损失却开始上升时，说明出现了过拟合，停止训练。实现上会使用四个大小为TEST_EPOCHS的队列，两个存放训练集损失，两个存放验证集损失。比如对训练集来说，一个存放1到TEST_EPOCHS的数据，称为队列1。一个存放TEST_EPOCHS + 1到2 * TEST_EPOCHS，称为队列2。每新训练完一个周期，得到当前的损失l_0，再从队列1中弹出一个数据l_1，从队列2中弹出一个数据l_2。于是，把l_0加入队列1中，并更新$Loss_1$：

$$Loss_1 = Loss_1 - l_1 + l_0$$

。把l_1加入加入队列2中，并更新$Loss_2$：

$$Loss_2 = Loss_2 - l_2 + l_1$$

。训练时会记录当前最小的验证集损失，并会把对应的模型存入磁盘。训练停止后，最小损失对应的模型成为最终模型。训练使用 DNNClassifier 的 train 方法，验证损失和准确率使用 evaluate 方法。神经网络使用的优化算法是 ProximalAdagradOptimizer，初始学习率设置为0.05，L2正则惩罚系数为0.0001。决策树使用 scikit-learn 中的决策树 DecisionTreeClassfier 。训练使用其 fit 方法。训练好的模型被保存在磁盘中。神

经网络模型直接通过设置 model_dir 参数保存，决策树模型使用 pickle[③] 保存。

预测阶段，对收集的失败测试用例覆盖的语句进行谓词预测。首先用Java的JDT提取语句中左值变量的特征，然后写入磁盘上的 csv 文件。Python程序读入 csv 文件中的特征和编码阶段存储的聚类模型。同时Python程序也会读入训练时使用的 csv 文件，以获取同样的 LabelEncoder 和 OneHotEncoder。使用聚类模型对特征的变量名、函数名和文件名划入某个已有的分类。然后用和训练时一样的 LabelEncoder 和 OneHotEncoder 对特征 FileName，MethodName，VarName，VarType，LastAssign，BodyUse 和 OutUse 进行编码。最后把预测变量的特征传入训练好的VAR模型和EXPR模型中预测。神经网络通过 predict 方法预测。该方法返回预测结果的一个列表。列表中一个元素对应于一个预测输入，该元素的 probabilities 域表示这个样本属于各个标签的概率， classes 域表示表示这个样本的预测标签。决策树通过 predict_proba 方法预测，得到一个矩阵。每行对应一个样本，每列对应这个样本属于某个标签的概率。 VAR模型会输出这个变量出现在谓词中的概率，EXPR模型会对一个样本的概率值排序，取概率最高的200个输出。最后，将同一个变量其VAR模型的输出P_{var}和EXPR模型的输出P_{pred_i}相乘，把联合概率大于0.005的变量及谓词输出。

验证阶段是在正常的缺陷定位中不存在的阶段。在测试中被用于测试模型的预测准确性。在验证阶段，输入数据会被使用 scikit-learn 的 train_test_split 函数随机分为60%的训练集，20%的验证集和20%的测试集。然后使用训练集和验证集去训练模型，对得到的模型使用测试集验证其准确率。神经网络的准确率计算使用 DNNClassifier 自带的 evaluate 方法的域 accuracy 获得。决策树的准确率使用 DecisionTreeClassfier 自带的 score 方法获得。而两者的其他评估数值由 scikit-learn 提供的计算接口获得。

过滤阶段，过滤掉预测出不合法或有副作用的谓词。使用JDT遍历谓词过滤。静态分析过滤后的谓词还会被单个编译过滤。每条语句最终只有过滤后的五个概率最高的谓词和它们的相反谓词。

5.2.2　生成预定义谓词模块

生成预定义谓词使用Java的JDT遍历抽象语法树。

对于一个返回语句，只要其返回值不是空 return;，就生成谓词。对于条件语句 if，for，while，do，switch，生成其条件表达式的谓词。对于赋值语句和变量声明语句，如果其左值是基本数据类型，获取当前行可见的其他同类型变量，生成谓词。

预定义生成的谓词数量往往大于预测谓词的数量。过滤的时候使用编译过滤速度较慢。其实预定义谓词并不需要过滤。首先，条件语句的谓词就是其条件表达式和其

③ https://docs.python.org/3/library/pickle.html

条件表达式取反，这种谓词肯定是编译正确的，所以不需要过滤。对于返回语句来说，只要验证 v > 0 是否编译正确就可以知道 v < 0 等谓词是否编译正确。对于数值对来说，只要验证 a > b 是否编译正确就可以知道 a < b 等谓词是否编译正确。因此减少过滤的谓词数量。

5.3 频谱信息收集模块

频谱信息收集模块主要分为两步，第一步是把谓词插入到代码中，第二步是执行测试用例收集执行信息。

插入谓词分为两种情况，一种是插入预测的谓词，一种是插入预定义的谓词。插入预测的谓词采用普通的插入方法，而插入预定义的谓词采用无副作用的插入方法。这两种方法都是使用JDT遍历抽象语法树，然后对函数声明 MethodDeclaration 节点进行一个对其字节点的递归遍历。这是因为插入谓词涉及到修改语法树，因此对于一个要被修改的节点来说，其父节点是被需要的甚至也是要修改的。

插入预测谓词时直接插入即可。而插入预定义谓词时，可能会涉及到新建变量、修改原有语句等。因为无副作用的插入方法会新建一个中间变量，然后用这个中间变量替换原有变量的位置。比如，为了在 return a.increase(); 处插入谓词 a.increase() > 0，首先是使用 VariableDeclarationFragment 新建一个变量，这个变量的变量名组成为 "automatic_" + lineNumber + "_" + varCount。其中 lineNumber 是行号， varCount 是一个递增的新变量计数。然后把这个 VariableDeclarationFragment 放入更高一层的 VarableDeclarationExpression，得到 int automatic_11_2。再将其放入 Assignment，得到 int automatic_11_2 = a.increase();。把原本的返回语句改为 return automatic_11_2;。然后谓词也要修改，原本要插入的谓词是 a.increase() > 0，现在要插入的谓词应该是 automatic_11_2 > 0。

为了收集执行信息，需要在代码中加入一些计数、标记、打印的语句。为了方便起见，这些方法被包装在一个负责收集这些信息的类的静态方法中，而这个类会被放入到缺陷代码的源代码中。

在代码被插入谓词和记录频谱的语句后，使用测试用例执行代码。频谱信息会被输出到文件中。频谱信息分为两类，第一类是SOBER算法的频谱信息。 SOBER算法的每个谓词会对应若干个覆盖信息。每一条覆盖信息对应一个测试用例的覆盖情况，包含三个数值，分别是谓词为真的次数、谓词为假的次数和当前测试用例是通过还是失败。第二类是除SOBER以外的公式的频谱信息。每个谓词对应一条覆盖信息，包含四个数值，分别是谓词真分支被覆盖的失败测试用例个数、谓词真分支被覆盖的通过测试用例个数、谓词（无论真假分支）被覆盖的失败测试用例个数和谓词（无论真假

分支）被覆盖的通过测试用例个数。

除了收集谓词的覆盖情况外，还会收集语句的频谱信息，包括语句被覆盖的失败测试用例个数和被覆盖的通过测试用例个数。

5.4　怀疑度计算模块

怀疑度计算模块根据频谱信息的不同也分为两种。一种是SOBER算法的怀疑度计算，一种是其他算法的怀疑度计算。在实现上，SOBER算法继承父类`TrueFalseCoverageNumberAlgori`抽象类，其他算法继承父类`PredicateCoverageAlgorithm`抽象类。子类需要实现两个函数，一个是`getName`返回算法名，一个是`getScore`返回怀疑度。父类负责从磁盘读取执行信息并处理为频谱信息，计算并结合语句的怀疑度和谓词怀疑度，根据怀疑度对语句进行排序，最后输出到磁盘。

在一条语句的多个谓词中，选择怀疑度最高的作为这个语句的最终谓词怀疑度。结合方式有MaxPred和LinPred。

第六章　实验与验证

6.1　研究问题

本文提出了一个结合了基于频谱的缺陷定位和基于状态覆盖的缺陷定位的缺陷定位框架，并且通过结合深入分析基于频谱的缺陷定位和基于状态覆盖的缺陷定位能够准确定位的原因。

本文试图回答以下几个问题：

1. **当使用不同的数据收集粒度时，缺陷定位技术的效果有什么不同？**
   这个研究问题探索了数据收集的粒度对缺陷定位结果的影响。最近研究使用的方法级别的缺陷定位从两个维度收集数据：语句级别和方法级别。但是没有现有方法探究了数据收集粒度对缺陷定位结果的影响。因为从不同的粒度收集数据会导致谓词数量的不同，而谓词数量对执行时间影响程度也不清楚。
2. **不同的怀疑度计算公式会如何影响缺陷定位的结果？**
   这个研究问题探索了缺陷定位公式的重要性，并分析了是否还能通过改进缺陷定位公式去提升缺陷定位效果，为后续研究提供了参考。
3. **缺陷定位技术的不同结合方式效果如何？**
   在章4.1中，本文探讨了结合缺陷定位技术的方法。这个研究问题研究了不同结合方式的效果。
4. **机器学习模型预测谓词的准确率如何？**
   这个研究问题探索了谓词预测模型的准确率，并且在不同的机器学习模型上进行了对比。
5. **哪种谓词能够更好地帮助定位？**
   这个研究问题比较了基于谓词预测的缺陷定位和本文提出的结合方法、现有方法之间的结果，分析谓词在定位缺陷中的作用和不足。
6. **和现有很多缺陷定位方法相比，本文的方法效果如何？**

6.2　实验数据

本文使用Defects4j数据集[21]（v1.0）作为实验对象，下文所提到的Defects4j都是v1.0版本。 Defects4j数据集含有357个真实的缺陷，来自五个大型的开源软件项目，见表6.1。其中，“实验缺陷数目”是本文在实验中实际使用了的缺陷数目，“代码行数（千行）”

项目名称	缺陷数目	实验缺陷数目	代码行数（千行）	测试用例数目
JFree**Chart**	26	26	96	2205
Closure compiler	133	122	90	7927
Apache commons-**Math**	106	98	85	3602
Apache commons-**Lang**	65	57	22	2245
Joda-**Time**	27	27	28	4130
总计	357	330	321	20109

表 6.1　Defects4j数据集缺陷情况

表示的是这五个项目最近的版本的代码行数。本文只使用了330个缺陷是因为其余27个缺陷因为插装后的代码太大导致“code too large"的报错而无法运行。

6.3　实验标准

为了验证本文提出的结合后的缺陷定位的效果，以及预测谓词的效果，需要一套衡量缺陷定位效果的标准和预测谓词效果的标准。

6.3.1　缺陷定位结果标准

本文使用两种常用的缺陷定位标准： Top-k 的召回率和 EXAM 分数。

Top-k 的召回率衡量的是有多少缺陷能够排在怀疑列表的前k位。根据Kochhar等人的研究[23]，超过73%的参与者认为观察怀疑列表的前5个程序元素是可以接受的，几乎所有参与者都认为10个程序元素是可以接受的最大的需要观察的程序元素个数。所以本文采用1，3，5，10作为k值。对于怀疑度分数相同的程序元素，它们的排名会使用平均排名，这也是很多已有研究使用的方法[36, 41, 46, 51]。

EXAM 分数衡量的是开发者需要查看多少个位置才能看到真正的缺陷。

$$\text{EXAM} = \frac{n}{N}$$

其中N表示候选的程序元素个数（比如被失败的测试用例覆盖的语句），n表示缺陷程序元素排在怀疑度列表的第n位。 EXAM 分数是一个0到1之间的值，且值越小越好。它反映出了含有缺陷的程序元素在所有可疑程序元素中的相对位置，体现出整个缺陷定位方法的效果。很多工作都使用了这个方法[36, 45]。

6.3.2　谓词预测结果标准

本文对谓词预测结果的评判方法采用机器学习分类问题的评判方法。

真实情况	预测结果	
	正例	反例
正例	TP（真正例）	FN（假反例）
反例	FP（假正例）	TN（真反例）

表 6.2　分类结果混淆矩阵

首先引入一些评估分类问题效果时需要的统计量。混淆矩阵统计量见表6.2。样本总数为N。正例反例为两个相对的概念。对于多分类问题，其余分类都为反例。

准确率（accuracy）衡量的是分类正确的样本数占样本总数的比例：

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

。

准确率能够衡量一个机器学习模型的预测情况，但是有的时候只有准确率还不够。有的时候人们还关心查准率和召回率。

查准率（precision）衡量的是被分类为正例的样本中有多少是真的正例：

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

。

查全率或召回率（recall）衡量的是正例中有多少被分类为正例：

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

。

查准率和查全率是一对相互矛盾的度量。一般来说，查准率高时，查全率低，而查准率低时，查全率高。 F1 度量结合查准率和查全率，给出一个便于比较的统一的值：

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

。

对于二分类问题来说，正例负例就是二分类的两个类别。但是对于多分类来说，每个类都可以是正例，这时其余类别就是负例。所以多分类问题的查准率，查全率和 F1 的对数和分类的类别数量一致。如果分类的类别数量非常多，观察庞大的查准率、查全率和 F1 并不能直观地描述预测结果，需要将各个分类的查准率、查全率和 F1 结合起来。 TP_i, FN_i, FP_i, TN_i表示多分类问题下，分类i的真正例、假反例、假正例和真

反例的数目。分类数记为C，c_i表示分类为i的样本数量。多分类问题的查准率、查全率和 F1 有以下几种结合方式：

- **micro**：
  先在全局的角度计算真正例、假反例和假正例的个数，然后带入公式计算查准率、查全率和 F1。以查准率为例：

$$\text{Precision} = \frac{\sum_{i=1}^{C} TP_i}{\sum_{i=1}^{C} (TP_i + FP_i)}$$

  。
- **macro**：
  先收集每个分类的真正例、假反例和假正例的个数，带入公式计算每个分类的查准率、查全率和 F1，最后求均值。这个不会考虑标签的不均衡。以查准率为例：

$$\text{Precision}_\text{i} = \frac{TP_i}{TP_i + FP_i}$$

$$\text{Precision} = \frac{1}{C} \sum_{i=1}^{c} \text{Precision}_\text{i}$$

  。
- **weighted**：
  先收集每个分类的真正例、假反例和假正例的个数，带入公式计算每个分类的查准率、查全率和 F1，最后根据分类的支持度（该分类的样本在总样本中所占的数目）分配权重求均值。这个算法与 macro 相比，考虑了标签的不均衡性。以查准率为例：

$$\text{Precision}_\text{i} = \frac{TP_i}{TP_i + FP_i}$$

$$\text{Precision} = \sum_{i=1}^{c} \frac{c_i}{C} \text{Precision}_\text{i}$$

  。

本文采用weighted的方式得到多分类问题的查准率、查全率和F1。

6.4 实验设置

6.4.1 数据收集粒度

本文考虑两种数据收集粒度：语句级别和函数级别。因为本文关注于函数的缺陷

定位，所以对于函数级别的数据收集，只需要直接在函数的出入口收集就可以。而对于语句级别的数据收集，则需要取一个函数内所有语句的最大怀疑度值作为这个函数的怀疑度。本文默认配置使用语句级别的数据收集。

6.4.2　怀疑度公式

怀疑度的公式上，本文实现了七种怀疑度计算的公式。五种是经典的基于频谱的缺陷定位公式见表3.3，两种是基于频谱的缺陷定位的公式3.2.1和公式3.2.2 {TODO:公式引用问题}。本文默认配置使用Ochiai作为怀疑度公式。

6.4.3　结合怀疑度的方法

在章4.1的讨论中，本文使用了公式

$$c(s,e,r) = max_{p \in s(e)} r(p)$$

计算得到了程序元素e的怀疑度，并提出了MaxPred和LinPred两种结合方式。

在本文的实验中，默认配置使用$\alpha = 0.5$的LinPred。其中，第一个怀疑度使用基于频谱的缺陷定位计算得到（$s_0(e)$产生的谓词恒为`true`），第二个使用基于状态覆盖的缺陷定位的谓词计算得到。

6.4.4　谓词

为了探索谓词在缺陷定位中的作用，本文把谓词分为三大类：预定义谓词、预测谓词和恒真谓词。对预定义谓词，又根据其定义分为三类：分支、返回和数值对。在本文的试验中，默认配置使用预定义谓词。

6.5　实验结果与分析

6.5.1　不同数据收集粒度对比

本小节试图通过探索不同数据收集粒度对缺陷定位效果效率的影响，来回答第一个研究问题。对于两种不同的数据收集粒度，我们都进行了实验。在语句级别收集数据的方法和此前的统计性调试的方法一样。而在函数级别收集数据的时候，谓词的类型会有差别。首先是没有“分支”类型的谓词。然后在收集“数值对”这样的预定义谓词时，不存在对应的赋值语句，所以转而收集所有可以访问的类成员变量和函数参

数的“数值对”。“数值对”类型的谓词会被插入在函数入口和出口处。“返回”类型的谓词会被插入在函数的出口处，如果函数有返回值的话。

实验结果如图6.1所示。图中LINPRED表示在语句级别收集数据， LINPRED-M表示在函数级别收集数据。根据实验结果，我们可以发现LINPRED的效果在Top-k（k=1,3,5,10）和 EXAM 分数上都优于LINPRED-M的效果。具体来说，在Top-1指标上，LINPRED相比于LINPRED-M提升了67.1%。

发现 1. 更细粒度的数据收集会让缺陷定位效果变好。

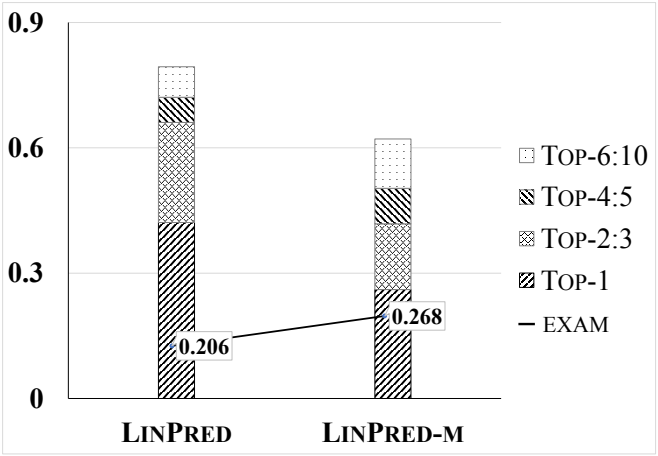

图 6.1　语句级别和函数级别数据收集的缺陷定位效果对比

数据收集粒度变细之后，谓词数量会增多。为了进一步探索增加谓词之后缺陷定位的效率是否会受到影响，本文比较了在语句级别收集数据和在函数级别收集数据的执行时间。我们发现，平均来说LINPRED的谓词数量大约是LINPRED-M谓词数量的4.1倍，但是LINPRED的执行时间只是LINPRED-M的1.4倍。同时，LINPRED的执行时间也是平均在三分钟以内。相比于提升数据收集粒度、降低谓词数量，在语句级别收集数据并不会严重地影响性能，同时能带来极大的效果提升。

6.5.2　不同的怀疑度计算公式

本小节将分析比较不同的怀疑度公式对缺陷定位结果的影响，并回答第二个研究问题。在章6.4.2中，我们使用了七个怀疑度公式。然而 SOBER 公式因为插装的内容较多，导致出现“code too large”的编译错误的项目较多，所以将 SOBER 公式排除。因此剩下五种基于频谱的缺陷定位公式和一种基于状态覆盖的缺陷定位公式。

实验结果如图6.2。图中展示的是每个怀疑度公式的 Top-k 和 EXAM 分数。图中的“SD”表示的是统计性调试的结果。从实验结果可以看出，前五个基于频谱的缺陷定位公式并没有太大的差别，这和之前的研究结果一致。统计性调试的结果则表现不

理想。可以发现基于频谱的缺陷定位公式的效果优于基于状态覆盖的缺陷定位公式效果。

发现 2.　基于频谱的缺陷定位公式的效果优于基于状态覆盖的缺陷定位公式效果

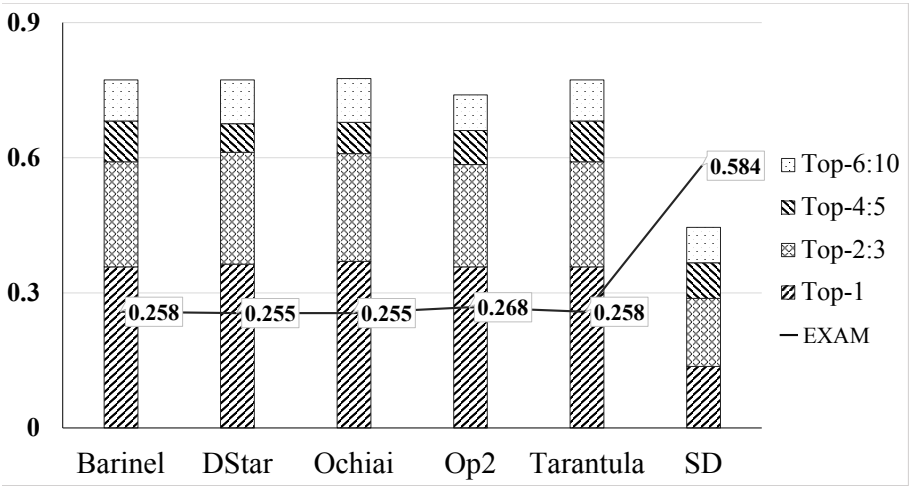

图 6.2　不同怀疑度公式的效果对比

为了深入分析基于状态覆盖的缺陷定位公式效果不好的原因，我们手动分析了统计性调试效果不好的缺陷。我们发现这是由于很多缺陷都只有一个失败的测试用例，这导致一个谓词的真分支往往只被一个失败的测试用例覆盖。这会导致统计性调试的公式中$t_f = 1$，导致$log(t_f) = 0$，然后$\frac{log(F)}{log(t_f)} = INF$，于是最终计算得到的怀疑度为0。这样谓词本身已经失去了甄别缺陷的能力，仅仅因为测试用例的数量就导致很多谓词的怀疑度为零。而统计性调试在此前的实验中有效是因为此前的实验对象中一个缺项往往对应多个失败的测试用例。比如在Liblit[26]的实验中，对每一个研究对象生成32000个随机输入。其测试用例数量远远大于 Defects4j 这样的实际项目。而基于频谱的缺陷定位公式却仍然能够在失败测试用例数量少的情况下得到比较靠谱的结果，因为它对失败测试用例数量的依赖没有基于状态覆盖的缺陷定位公式的高。统计性调试公式仍然有它的应用场景。在使用缺陷定位公式时，可以根据其应用场景选择合适的公式。

6.5.3　不同的结合方式

本小节将深入探讨不同的结合方式，来回答第三个研究问题。结合的方式被分为了两种MaxPred和LinPred。图6.3展示了不同结合方式的 Top-k 和 EXAM 结果。“SBFL”表示传统的基于频谱的缺陷定位的结果（谓词恒为真），这里的MaxPred的谓词使用预定义谓词。

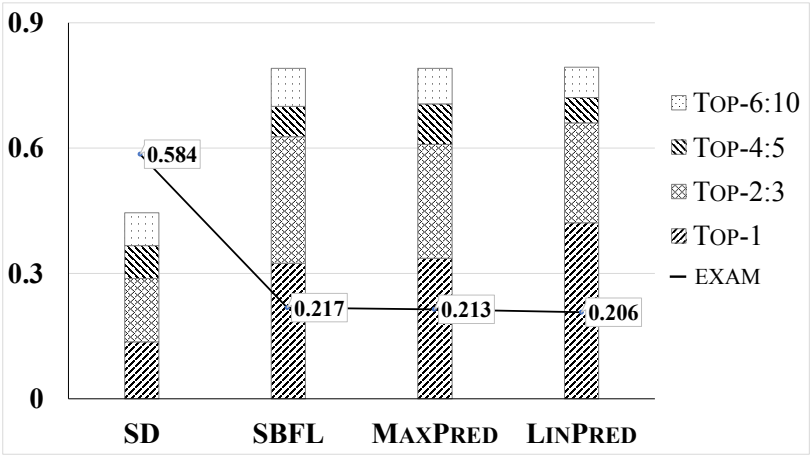

图 6.3　不同的结合方式的对比

从图中可以看出，结合后的MaxPred和LinPred在 Top-1 上优于原始的方法。LinPred不仅获得了最好的 Top-k（k=1,3,5,10）值，还拥有最好的 EXAM 值。LinPred在 Top-1 上与传统的统计性调试比提升了208.9%，与基于频谱的缺陷定位比提升了29.9%。可见 SBFL 和 MaxPred 结合之后的LinPred效果非常好。

图6.3中的LinPred采用的默认的$\alpha = 0.5$。为了分析 SBFL 和 MaxPred 哪一种对结果的影响更大，采用不同的α值进行实现，结果如图6.4。

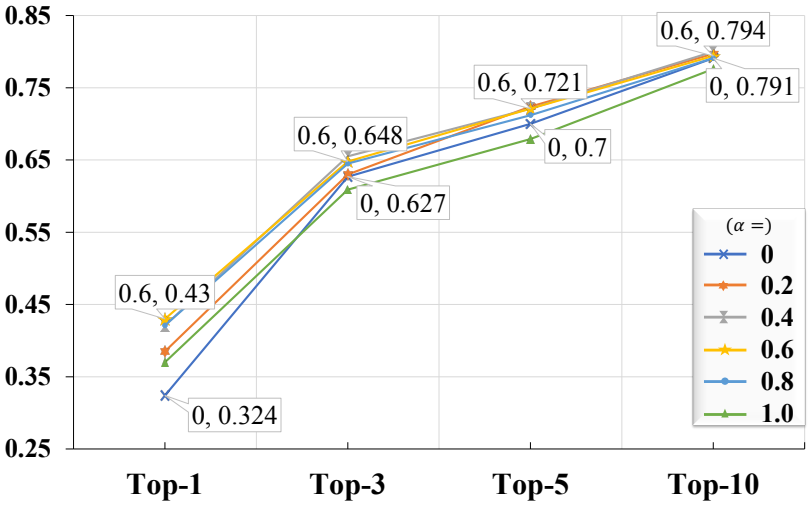

图 6.4　LinPred中α取不同值时，其 Top-k 值的变化情况

从图中可以看出，LinPred总是比原始的 SBFL 方法（$\alpha = 0$）在 Top-1 上的效果好。在 Top-k（k=3,5,10）上，MaxPred的效果一直垫底，可见MaxPred虽然能给出比较不错的分数，但是单一的MaxPred无法突出缺陷的语句，需要 SBFL 和它互补。当α落在[0.4-0.8]之间的时候，LinPred的效果最好。不过总体上来说α值的影响不会太大。

图6.5展示了 MaxPred，LinPred，基于频谱的缺陷定位和基于状态覆盖的缺陷定位在 Defects4j 上各个项目、各个公式上 Top-1 的结果。可以看到LinPred在各个项目、各个公式上都有比较好的效果。

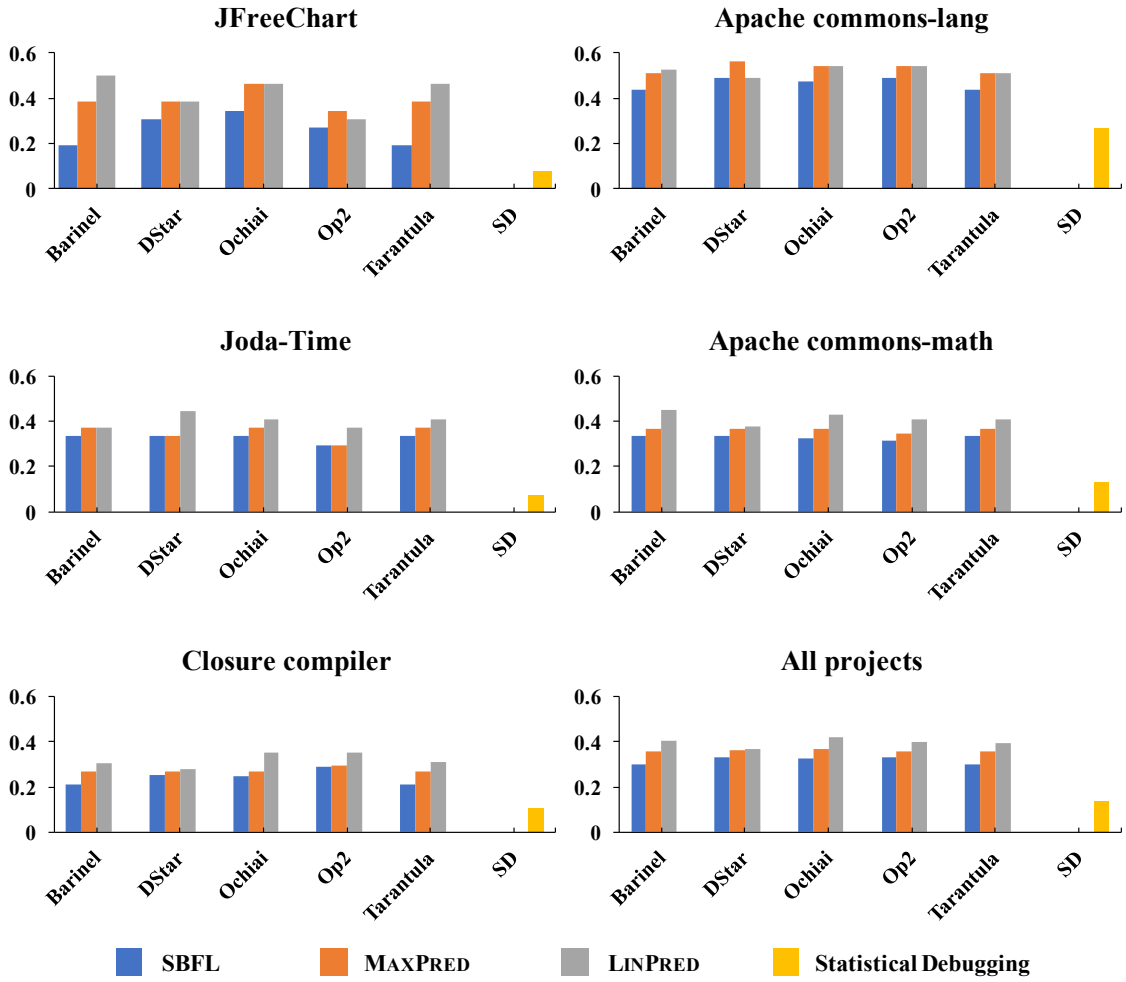

图 6.5　MaxPred、LinPred和传统的基于频谱和基于状态覆盖的缺陷定位在各个项目、各个公式上的 Top-1 结果对比

6.5.4　基于机器学习的谓词预测模型效果

本小节将对基于神经网络的谓词预测模型和基于决策树的谓词预测模型的预测效果进行研究，并回答第四个研究问题。

先考虑VAR模型。 VAR模型的结果如表6.3所示。

再分析EXPR模型。 EXPR模型的预测结果如表6.4。

{TODO:...}

项目	神经网络模型				决策树模型			
	准确率	查准率	查全率	F1分数	准确率	查准率	查全率	F1分数
Math	85.31%	85.46%	85.31%	84.69%	88.64%	88.52%	88.64%	88.56%
	89.55%	90.29%	89.55%	89.13%	98.43%	98.43%	98.43%	98.42%
Chart	89.00%	89.05%	89.00%	88.25%	92.75%	92.70%	92.75%	92.72%
	90.82%	91.28%	90.82%	90.17%	98.30%	98.29%	98.30%	98.29%
Lang	83.86%	83.85%	83.86%	83.46%	84.75%	84.64%	84.75%	84.67%
	90.13%	90.50%	90.13%	89.90%	96.81%	96.83%	96.81%	96.79%
Time	84.20%	84.30%	84.20%	83.74%	86.29%	86.24%	86.29%	86.25%
	90.34%	90.84%	90.34%	90.04%	98.66%	98.67%	98.66%	98.65%
Closure	79.15%	79.12%	79.15%	78.10%	83.85%	83.55%	83.85%	83.67%
	85.23%	86.31%	85.23%	84.57%	97.68%	97.68%	97.68%	97.66%
全部	82.93%	82.98%	82.93%	82.20%	86.27%	86.09%	86.27%	86.16%
	88.20%	88.96%	88.20%	87.72%	97.86%	97.87%	97.86%	97.85%

表 6.3　VAR模型的预测效果，每个项目第一行为测试集结果，第二行为训练集结果

项目	神经网络模型				决策树模型			
	准确率	查准率	查全率	F1分数	准确率	查准率	查全率	F1分数
Math	26.01%	20.69%	26.01%	21.13%	50.15%	49.05%	50.15%	47.72%
	34.19%	27.02%	34.19%	27.59%	84.90%	84.19%	84.90%	82.86%
Chart	39.42%	37.33%	39.42%	37.02%	49.82%	49.02%	49.82%	48.43%
	47.48%	44.06%	47.48%	43.79%	84.60%	84.23%	84.60%	82.94%
Lang	42.67%	33.42%	42.67%	36.10%	57.12%	55.57%	57.12%	55.04%
	49.92%	38.35%	49.92%	41.63%	86.50%	84.85%	86.50%	84.20%
Time	43.74%	33.31%	43.74%	36.87%	64.69%	63.86%	64.69%	63.07%
	48.72%	36.08%	48.72%	40.14%	94.34%	93.58%	94.34%	93.30%
Closure	24.76%	19.47%	24.76%	20.98%	34.12%	32.44%	34.12%	32.26%
	33.04%	25.43%	33.04%	27.40%	86.76%	86.03%	86.76%	84.92%
全部	30.90%	24.72%	30.90%	26.15%	46.52%	45.17%	46.52%	44.50%
	38.69%	30.41%	38.69%	32.20%	86.57%	85.71%	86.57%	84.67%

表 6.4　EXPR模型的预测效果，每个项目第一行为测试集结果，第二行为训练集结果

谓词类型	项目					
	Chart	Lang	Time	Math	Closure	总计
预定义谓词	913.6	119.5	1064.2	1973.0	2426.8	1662.8
预测谓词	-	-	-	-	-	-
恒真谓词	566.5	75.8	892.7	278.3	3401.3	1470.9

表 6.5　平均每个项目的谓词数量

6.5.5　不同谓词对结果的影响

在之前的研究问题的分析中，我们已经发现了LinPred具有非常好的效果。相比于基于频谱的缺陷定位，LinPred利用了谓词进行进一步的状态划分。在本小节，我们将会深入探究谓词在缺陷定位中起的作用，来回答第五个研究问题。

谓词在项目中的平均数量如表6.5所示。 Closure 因为项目较大所以谓词数量较多。

首先分析谓词对定位效果的影响。对于程序元素e，一个谓词被认为是对定位效果有积极的影响，如果$c_0 > c_1$且e是缺陷程序元素，或者$c_0 < c_1$且e不是缺陷程序元素。其中c_0表示使用预定义谓词的MaxPred的结果， c_1表示基于频谱的缺陷定位的结果（谓词恒真的MaxPred的结果）。一个谓词被认为是对定位效果有消极影响，如果$c_0 < c_1$且e是缺陷程序元素，或者$c_0 > c_1$且e不是缺陷程序元素。于是谓词被分为两类，这两类在不同的怀疑度计算公式下的分布情况如图6.6。可以发现大部分谓词还是对结果有积极影响的，只有少部分会对结果产生不好的影响。

{TODO:机器学习预测的谓词}

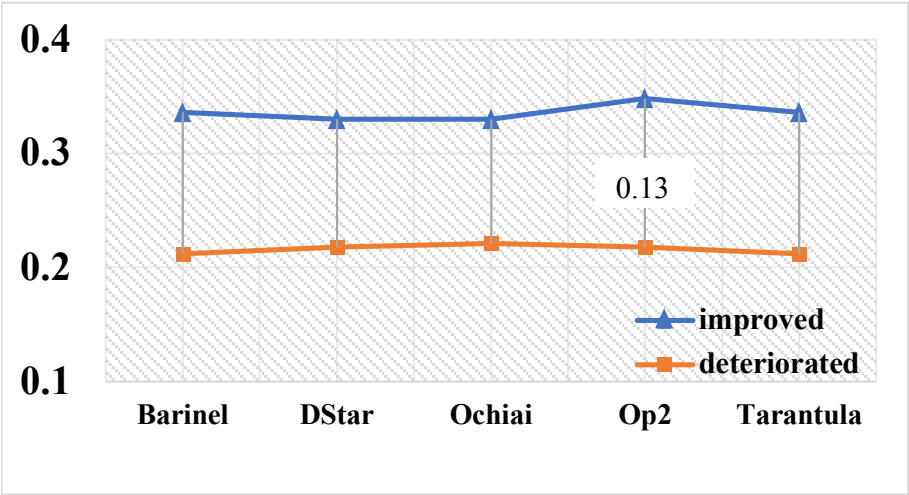

图 6.6　预定义谓词在缺陷定位中的作用

为了探索是三种预定义谓词中的哪种谓词起了关键作用，我们把三种预定义谓词分别作为谓词集合，进行实验。实验结果如图6.7所示。 LinBrn表示只使用分支谓词，

LinRet表示只使用返回值谓词，LinSca表示只使用数值对谓词。可以发现，LinBrn拥有比LinRet和LinSca更好的 Top-k（k=1,3,5,10）和 EXAM 值。所以分支谓词的效果最明显。 LinBrn在 Top-1 上的效果比基于频谱的缺陷定位效果还要好，不过基于频谱的缺陷定位在其他 Top-k 上效果更好，说明了二者的互补性。

发现 3. *在预定义谓词中，分支谓词对LinPred效果提升最明显。*

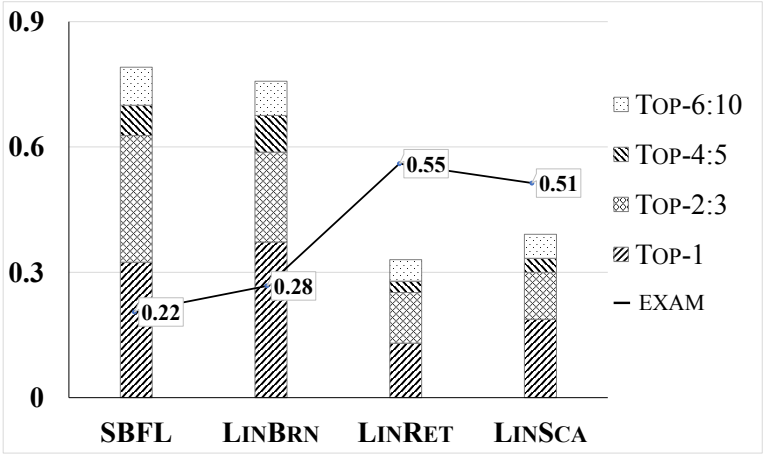

图 6.7　使用不同的谓词的缺陷定位效果对比

为了进一步验证我们的发现，我们比较了使用三种预定义谓词的MaxPred和只使用了分支谓词的MaxPred（称为MaxBranch），结果如图6.8。可以发现MaxPred和MaxBranch的结果非常相近，虽然MaxBranch少了两种谓词，但是核心的分支谓词起了很大的作用。

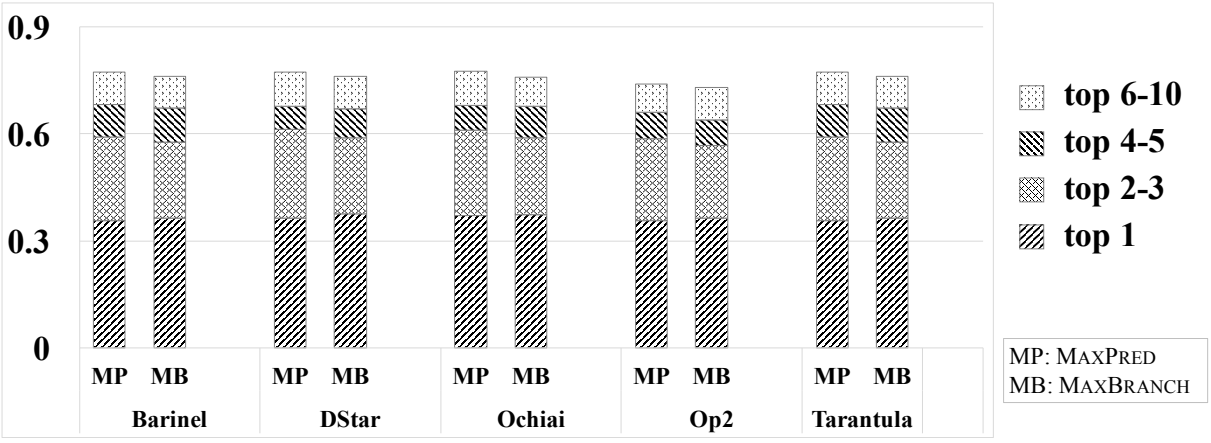

图 6.8　使用不同的谓词的缺陷定位效果对比

第七章　总结与未来工作

本章对本文的工作进行总结，并且对未来的研究方向进行展望。

7.1　总结

本文结合了现有的缺陷定位技术，提出了结合现有缺陷定位技术的新的缺陷定位技术。具体来说，

1. 本文提出了一种结合基于频谱的缺陷定位和基于状态覆盖的缺陷定位的方法LinPred，并且在实际数据集 Defects4j 中与传统的基于频谱的缺陷定位和基于状态覆盖的缺陷定位进行了比较。深入分析了基于频谱的缺陷定位和基于状态覆盖的缺陷定位起作用的原因，并且将其互补地结合起来。可以发现LinPred在各个项目、各个公式上都有比较好的效果。
2. 本文提出了一种基于机器学习的预测谓词模型，来替代基于装填覆盖的缺陷定位中的预定义谓词。{TODO:...}

本文发现，在LinPred方法下：

1. 更细粒度的数据收集会让缺陷定位效果变好。
2. 在Defects4j数据集中基于频谱的缺陷定位公式的效果优于基于状态覆盖的缺陷定位公式效果。
3. 在预定义谓词中，分支谓词对LinPred效果提升最明显。本质上与使用分支覆盖的基于频谱的缺陷定位技术相同。

7.2　未来工作

参考文献

[1] R Abreu, P Zoeteweij and A. J. C Van Gemund. "*An Evaluation of Similarity Coefficients for Software Fault Localization*". In: *Pacific Rim International Symposium on Dependable Computing*, **2006**: 39–46.

[2] R Abreu, P Zoeteweij and A. J. C Van Gemund. "*On the Accuracy of Spectrum-based Fault Localization*". In: *Testing: Academic and Industrial Conference Practice and Research Techniques - Mutation*, **2007**: 89–98.

[3] Rui Abreu, Peter Zoeteweij and Arjan J. C Van Gemund. "*Spectrum-Based Multiple Fault Localization*". In: *Ieee/acm International Conference on Automated Software Engineering*, **2009**: 88–99.

[4] Hiralal Agrawal, Richard A. Demillo and Eugene H. Spafford. *Debugging with dynamic slicing and backtracking*, **1993**: 589–616.

[5] Elton Alves, Milos Gligoric, Vilas Jagannath *et al.* "*Fault-localization using dynamic slicing and change impact analysis*". In: *Ieee/acm International Conference on Automated Software Engineering*, **2011**: 520–523.

[6] Thomas Ball, Mayur Naik and Sriram K Rajamani. "*From symptom to cause: localizing errors in counterexample traces*". *Acm Sigplan Notices*, **2003**, *38*(1): 97–105.

[7] Richard Ernest Bellman. *Dynamic programming*. Princeton University Press, **1957**.

[8] Satish Chandra, Emina Torlak, Shaon Barman *et al.* "*Angelic debugging*". In: *International Conference on Software Engineering*, **2011**: 121–130.

[9] Mike Y. Chen, Emre Kiciman, Eugene Fratkin *et al.* "*Pinpoint: Problem Determination in Large, Dynamic Internet Services*". In: *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, **2002**: 595–604.

[10] James S. Collofello and Scott N. Woodfield. *Evaluating the effectiveness of reliability-assurance techniques*. Elsevier Science Inc., **1989**: 191–195.

[11] Valentin Dallmeier, Christian Lindig and Andreas Zeller. *Lightweight Defect Localization for Java.* Springer Berlin Heidelberg, **2005**: 528–550.

[12] Richard A. Demillo, Hsin Pan and Eugene H. Spafford. "*Critical slicing for software fault localization*". In: **1996**: 121–134.

[13] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo *et al.* "*The Daikon system for dynamic detection of likely invariants*". *Science of Computer Programming*, **2007**, *69*(1-3): 35–45.

[14] Alex Groce, Daniel Kroening and Flavio Lerda. "*Understanding Counterexamples with explain*". *In Computer-Aided Verification*, **2004**, *3114*: 453–456.

[15] Gyim, Tibor Thy, Besz *et al.* "*An efficient relevant slicing method for debugging*". *Acm Sigsoft Software Engineering Notes*, **1999**, *24*(6): 303–321.

[16] Mary Jean Harrold, Gregg Rothermel, Kent Sayre *et al.* "*An empirical investigation of the relationship between spectra differences and regression faults*". *Software Testing Verification & Reliability*, **2000**, *10*(3): 171–194.

[17] Monica Hutchins, Herb Foster, Tarak Goradia *et al.* "*Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria.*" In: *international Conference on Software Engineering*, **1994**: 191–200.

[18] James A. Jones and Mary Jean Harrold. "*Empirical evaluation of the tarantula automatic fault-localization technique*". In: *Ieee/acm International Conference on Automated Software Engineering*, **2005**: 273–282.

[19] Jones, A James, Harrold *et al.* "*Visualization of test information to assist fault localization*". *Biochemical Engineering Journal*, **2002**, *24*(2): 115–123.

[20] Xiaolin Ju, Shujuan Jiang, Xiang Chen *et al.* "*HSFal: Effective fault localization using hybrid spectrum of full slices and execution slices*". *Journal of Systems & Software*, **2014**, *90*(1): 3–17.

[21] René Just, Darioush Jalali and Michael D. Ernst. "*Defects4J: a database of existing faults to enable controlled testing studies for Java programs*". In: *International Symposium on Software Testing and Analysis*, **2014**: 437–440.

[22] Z. A. Al-Khanjari, M. R. Woodward, Haider Ali Ramadhan *et al.* "*The Efficiency of Critical Slicing in Fault Localization*". *Software Quality Journal*, **2005**, *13*(2): 129–153.

[23] Pavneet Singh Kochhar, Xin Xia, David Lo *et al.* "*Practitioners' expectations on automated fault localization*". In: *International Symposium on Software Testing and Analysis*, **2016**: 165–176.

[24] Bogdan Korel and Janusz Laski. "*Dynamic program slicing ☆*". *Information Processing Letters*, **1988**, *29*(3): 155–163.

[25] Tien Duy B. Le, David Lo, Claire Le Goues *et al.* "*A learning-to-rank based fault localization approach using likely invariants*". In: *International Symposium on Software Testing and Analysis*, **2016**: 177–188.

[26] Ben Liblit, Mayur Naik, Alice X. Zheng *et al.* "*Scalable statistical bug isolation*". In: **2005**: 15–26.

[27] Chao Liu, Long Fei, Xifeng Yan *et al.* "*Statistical Debugging: A Hypothesis Testing-Based Approach*". *IEEE Transactions on Software Engineering*, **2006**, *32*(10): 831–848.

[28] Chao Liu, Xifeng Yan, Long Fei *et al.* "*SOBER: statistical model-based bug localization*". In: *European Software Engineering Conference Held Jointly with ACM Sigsoft International Symposium on Foundations of Software Engineering*, **2005**: 286–295.

[29] Chao Liu, Xiangyu Zhang, Jiawei Han *et al.* "*Indexing Noncrashing Failures: A Dynamic Program Slicing-Based Approach*". In: *IEEE International Conference on Software Maintenance*, **2007**: 455–464.

[30] Xiaoguang Mao, Yan Lei, Ziying Dai *et al.* "*Slice-based statistical fault localization ☆*". *Journal of Systems & Software*, **2014**, *89*(1): 51–62.

[31] Meyer, Andréia Da Silvagarcia, Antonio Augusto Francosouza *et al.* "*Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (Zea mays L)*". *Genetics & Molecular Biology*, **2004**, *27*(1): 83–91.

[32] Seokhyeon Moon, Yunho Kim, Moonzoo Kim *et al.* "*Ask the Mutants: Mutating Faulty Programs for Fault Localization*". In: *IEEE Seventh International Conference on Software Testing, Verification and Validation*, **2014**: 153–162.

[33] Lee Naish, Jie Lee Hua and Kotagiri Ramamohanarao. "*A model for spectra-based software diagnosis*". *Acm Transactions on Software Engineering & Methodology*, **2011**, *20*(3): 1–32.

[34] Mike Papadakis and Yves Le Traon. *Metallaxis-FL: mutation-based fault localization*. John Wiley and Sons Ltd., **2015**: 605–628.

[35] Chris Parnin and Alessandro Orso. "*Are automated debugging techniques actually helping programmers?*" In: *International Symposium on Software Testing and Analysis*, **2011**: 199–209.

[36] Spencer Pearson, Jose Campos, Rene Just *et al.* "*Evaluating and Improving Fault Localization*". In: *International Conference on Software Engineering*, **2017**: 609–620.

[37] M Renieres and S. P Reiss. "*Fault localization with nearest neighbor queries*". In: *IEEE International Conference on Automated Software Engineering, 2003. Proceedings*, **2003**: 30–39.

[38] NIST Rep. *The Economic Impacts of Inadequate Infrastructure for Software Testing*, **2002**. `http://www.abeacha.com/NIST_press_release_bugs_cost.htm`, retrieved on 2018-04-18.

[39] Thomas Reps, Thomas Ball, Manuvir Das *et al.* "*The use of program profiling for software maintenance with applications to the year 2000 problem*". *Acm Sigsoft Software Engineering Notes*, **1997**, *22*(6): 432–449.

[40] Ehud Y. Shapiro. "*Algorithmic Program DeBugging*". **1982**.

[41] Friedrich Steimann, Marcus Frenkel and Abreu Rui. "*Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators*". In: *International Symposium on Software Testing and Analysis*, **2013**: 314–324.

[42] Mark Weiser. "*Program Slicing*". *IEEE Transactions on Software Engineering*, **1984**, *SE-10*(4): 352–357.

[43] Mark Weiser. "*Program slicing*". In: *International Conference on Software Engineering*, **1981**: 439–449.

[44] W. Eric Wong, Vidroha Debroy, Ruizhi Gao *et al.* "*The DStar Method for Effective Software Fault Localization*". *IEEE Transactions on Reliability*, **2014**, *63*(1): 290–308.

[45] W. Eric Wong, Vidroha Debroy, Richard Golden *et al.* "*Effective Software Fault Localization Using an RBF Neural Network*". *IEEE Transactions on Reliability*, **2012**, *61*(1): 149–169.

[46] W. Eric Wong, Ruizhi Gao, Yihao Li *et al.* "*A Survey on Software Fault Localization*". *IEEE Transactions on Software Engineering*, **2016**, *42*(8): 707–740.

[47] W. ERIC WONG and YU QI. "*BP NEURAL NETWORK-BASED EFFECTIVE FAULT LOCALIZATION*". *International Journal of Software Engineering & Knowledge Engineering*, **2009**, *19*(04): 573–597.

[48] Franz Wotawa. "*Fault Localization Based on Dynamic Slicing and Hitting-Set Computation.*" In: *International Conference on Quality Software*, **2010**: 161–170.

[49] Xiaoyuan Xie, Tsong Yueh Chen, Fei Ching Kuo *et al.* "*A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization*". *Acm Transactions on Software Engineering & Methodology*, **2013**, *22*(4): 31.

[50] Yingfei Xiong. *Fault Localization*, **2018**. `http://sei.pku.edu.cn/~xiongyf04/SA/2017/18_fault_localization.pdf`, retrieved on 2018-04-07.

[51] Jifeng Xuan and Martin Monperrus. "*Learning to Combine Multiple Ranking Metrics for Fault Localization*". In: *IEEE International Conference on Software Maintenance and Evolution*, **2014**: 191–200.

[52] A. Zeller and R. Hildebrandt. "*Simplifying and Isolating Failure-Inducing Input*". *Software Engineering IEEE Transactions on*, **2002**, *28*(2): 183–200.

[53] Andreas Zeller. "*Isolating cause-effect chains from computer programs*". In: *ACM Sigsoft Symposium on Foundations of Software Engineering*, **2002**: 1–10.

[54] Xiangyu Zhang, Neelam Gupta and Rajiv Gupta. "*Locating faults through automated predicate switching*". **2006**, *2006*: 272–281.

[55] Xiangyu Zhang, Neelam Gupta and Rajiv Gupta. "*Pruning dynamic slices with confidence*". *Acm Sigplan Notices*, **2006**, *41*(6): 169–180.

[56] Xiangyu Zhang, R Gupta and Youtao Zhang. "*Precise dynamic slicing algorithms*". In: *International Conference on Software Engineering, 2003. Proceedings*, **2003**: 319–329.

[57] Daming Zou, Ran Wang, Yingfei Xiong *et al.* "*A genetic algorithm for detecting significant floating-point inaccuracies*". In: *Ieee/acm IEEE International Conference on Software Engineering*, **2015**: 529–539.

附录 A　附件

致谢

感谢北京大学软件工程所，让我能够接触到前沿的科研项目。浓厚的学术氛围感染了我，敦促我不断学习，打下了科研的坚实基础。

感谢熊英飞研究员，张路教授和郝丹副教授对我的指导。我从大三开始就在张路老师的小组里学习，近五年的时间里三位老师对我耐心地指导和帮助，让我受益颇多。我也在熊英飞研究和张路教授的指导下发表了两篇CCF-A类的论文，一篇第一作者，一篇第二作者。

感谢姜佳君同学，和我一起讨论、完成这个研究。他提出了许多宝贵的想法与建议，并且与我一起实现了LinPred。

感谢王博同学和臧琳飞师姐，他们的基于机器学习的缺陷修复给了本文非常多的帮助。

最后，感谢我的父母一直陪伴着我、支持着我。他们一直是我坚强的后盾。

北京大学学位论文原创性声明和使用授权说明

原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律结果由本人承担。

论文作者签名：　　日期：　　年　　月　　日

学位论文使用授权说明

（必须装订在提交学校图书馆的印刷本）

本人完全了解北京大学关于收集、保存、使用学位论文的规定，即：

- 按照学校要求提交学位论文的印刷本和电子版本；
- 学校有权保存学位论文的印刷本和电子版，并提供目录检索与阅览服务，在校园网上提供服务；
- 学校可以采用影印、缩印、数字化或其它复制手段保存论文；
- 因某种特殊原因需要延迟发布学位论文电子版，授权学校在□一年/□两年/□三年以后在校园网上全文发布。

（保密论文在解密后遵守此规定）

论文作者签名：　　导师签名：　　日期：　　年　　月　　日