南京大學

研 究 生 毕 业 论 文

（申请工程硕士学位）

论 文 题 目 智能化源码警告分类系统的设计与实现

作 者 姓 名 葛宇

学科、专业名称 工程硕士（软件工程领域）

研 究 方 向 软件工程

指 导 教 师 陈振宇教授，房春荣助理研究员

2022 年 **5** 月 **16** 日

学　　号：**MF20320049**

论文答辩日期：**xxxx** 年 **xx** 月 **xx** 日

指 导 教 师：　　（签字）

Design and Implementation of Intelligent Source Code Alarm Classification System

by

Yu Ge

Supervised by

Professor Zhenyu Chen, Associate Professor Chunrong Fang

A dissertation submitted to
the graduate school of Nanjing University
in partial fulfilment of the requirements for the degree of
Master of Engineering
in
Software Engineering

Software Institute
Nanjing University

May 16, 2022

南京大学研究生毕业论文中文摘要首页用纸

毕业论文题目：智能化源码警告分类系统的设计与实现

工程硕士（软件工程领域）专业 2020 级硕士生姓名：葛宇

指导教师（姓名、职称）：陈振宇教授，房春荣助理研究员

摘　要

源码静态扫描工具因为能够快速地检测出项目中的代码缺陷而受到开发者的广泛使用。由于源码静态扫描工具是在编译时执行扫描的，因此往往过度估计了程序的执行路径，导致产生了大量的误报警告。为了找到正报警告，开发者需要耗费很多时间去手动检测警告。当前，为了减轻开发者审计的压力，往往采用机器学习技术来检测正报警告。然而这类方法还存在一些不足之处：其一，目前的研究大都采用合成测试数据集。该类型数据集不能代表真实世界的项目；其二，很多方法把警告代码视为自然语言文本来学习分类模型，这会遗漏代码的重要语法信息。部分方法使用手工特征学习模型，虽然提取了警告某个方面的特征，但是缺少警告代码中包含的语法特征。

针对上述不足之处，本文首先在 5 个开源项目的多个 commit 版本上，使用差分分析技术构建了一个大规模的真实警告数据集，总共收集了 60404 个警告，其中正报 8483 个，误报 51921 个；其次搭建了一个基于抽象语法树的神经网络来表示警告代码，使用双向循环神经网络模型学习警告语句间的联系，生成警告代码的向量表示；接着提取 22 个警告手工特征，捕获代码中难以发现的警告度量信息并与代码特征融合；最后在警告融合特征上训练警告分类器模型。该警告分类器模型被用来对待测警告做出分类，以达到警告检测的目的。

为了评估本文提出的方法的有效性，本文与五种警告分类方法进行比较。实验结果显示本方法在 F1 值上最少提升了 13.00%。这证明本文提出的方法能更有效地检测误报警告，大大提升了警告检测的效率。

关键词： 误报警告检测；机器学习；警告数据集构建；特征融合

南京大学研究生毕业论文英文摘要首页用纸

THESIS: Design and Implementation of Intelligent Source Code Alarm Classification System

SPECIALIZATION: Software Engineering

POSTGRADUATE: Yu Ge

MENTOR: Professor Zhenyu Chen, Associate Professor Chunrong Fang

Abstract

Source code static scanning tools(SCSCTs) have been widely used by developers because they can quickly and easily detect code defects in projects. however, due to overestimating the execution path of the program when performing scanning at compile-time, Using SCSCTs can result in a large number of false-positive alarms. In order to find true-positive alarms, developers need to spend a lot of time manually detecting all alarms reported by SCSCTs. To relieve the pressure of developer auditing, the existing approaches often employ machine learning techniques to identify true-positive alarms. The approach is to apply machine learning to learn an alarm classifier on a set of hand-crafted features or code features of alarms. However, there are still some shortcomings in the existing ML-based alarm identification approache. First, most of the current research uses synthetic test datasets. such datasets have a large amount of data, but the injected code defects are artificially introduced, which cannot represent real-world projects and are not generalizable. Therefore, the training of the alarm classifier lacks a large number of real alarm data sets. Sencond, many approaches treat the alarm code as natural language text to learn the classification model and process the text into a token sequence or token word bag, which misses the important syntax information. Some approaches use hand-crafted features to learn a model. Although features of a certain aspect of the alarm are extracted, the lack of syntactic features contained in the alarm code cannot fully represent a warning. Therefore alarms lack a comprehensive feature in terms of feature representation.

This paper uses differential analysis technology to construct a large-scale, real-world alarm dataset on multiple commit versions of the collected five open-source

projects. 60,404 alarms are collected in all, including 8,483 true-positive alarms and 51,921 false-positive alarms. Next, an abstract syntax tree-based neural network is built to represent the alarm code, the entire abstract syntax tree is split into a series of sentence trees, and the sentence trees are encoded into vectors by capturing the lexical and syntactic knowledge of the sentences. Based on the sequence of sentence vectors, a bidirectional RNN model is used to learn the naturalness between sentences, and a vector representation of the alarm code is generated. Finally, the code features and software metrics features are combined, and machine learning techniques are used to train the alarm classifier model.

To evaluate our approach, we compare it with five alarm classification methods. The experimental results show that our approach improves by at least 13.00% in terms of F1. This proves that the method proposed in this paper can detect false-positive alarms more effectively. Besides, The source code alarm classifier constructed in this paper greatly improves the performance of false-positive alarm detection, reduces the review burden of developers, and fundamentally guarantees the availability of source code static scanning tools.

keywords: False-positive alarm detection, machine learning, alarm dataset construction, feature fusion

目　　录

插图清单

附表清单

第一章　引言

1.1　项目背景与意义

随着软件程序的规模和复杂性的上升，开发者引入的代码缺陷也随之变多，随之而来地，开发者往往需要耗费很多时间和精力来发现并去除代码缺陷。源码静态扫描工具在不用运行源码的情况下能够自动地揭露代码中的性能、安全、违反代码标准等缺陷 [1]。当前，研究者们提出 Fortify[①]、FindBugs[②]、Checkstyle[③]、CppCheck[④]、FlawFinders[⑤]等多种源码静态扫描工具，它们能够检测 C/C++、Java、Python、Javascript 等主流语言开发的程序。此外，源码静态扫描工具能够扫描整个项目的源代码，因此可以达到较好的测试覆盖率。源码静态扫描工具已经被广泛地应用到了商业软件的缺陷检测中，同时也已经成为了软件生命周期中不可或缺的重要组成成分 [2]。

然而，源码静态扫描工具会产生过多的误报。误报是指源码静态扫描工具在扫描过程中将一些没有代码缺陷的源码错误地标记为警告 [3]。尽管源码静态扫描工具使用的静态分析技术，在空间足够的条件下可以做到对所有可能的运行路径进行建模，能够规避掉很大一部分的误报，但在实际工作中可能会遇到因路径爆炸导致空间不足的情况出现。为了处理规模庞大的程序，避免因路径爆炸导致的空间不足问题，源码静态扫描工具采取过近似的分析方法，以牺牲精度来获得更好的可扩展性和速率 [4]。因此，在工业界中普遍使用忽略路径条件并假设所有路径都可行的路径不敏感分析，这必然会引入大量的误报 [5]。过多的误报加重了人工审核任务，直接阻碍了开发人员使用源码静态扫描工具的积极性。

目前已经有很多研究在减少误报上取得了一定的进展 [6]。在诸多误报检测研究中，最普遍的做法是在一组警告的特征上，应用机器学习训练警告分类器模型，以此来过滤误报警告 [7]。但是已有研究在警告分类器的构建上存在如

[①] https://www.joinfortify.com/
[②] http://findbugs.sourceforge.net/
[③] https://checkstyle.sourceforge.io/
[④] http://cppcheck.net/
[⑤] https://formulae.brew.sh/formula/flawfinder

下两个问题：第一，缺少大型的，真实警告数据集。由于警告表现出多样化和复杂的行为，训练警告分类器需要有真实警告和虚假警告代码示例的大型标记数据集。这对于 CNN、RNN、GNN 等高级神经网络模型尤其重要 [8]。然而，用于训练警告分类器模型的现有数据集具有很多局限性，例如有限的警告上下文以及合成和人为构造的源代码；第二，现有的方法没有总结出一个全面的特征以表征警告。传统的警告代码特征提取方法大都将警告代码视为自然语言文本，将其处理成 token 序列或者 token 词袋，这会遗漏警告的语法结构和语义信息 [9]。近些年的研究仅仅局限于软件度量、历史信息、源码本身其中一个点做特征提取，而且基于每个点提取的特征也不尽相同。由此可见，当前方法提取的警告特征角度较为单一。

基于上述背景，本文构建了一个警告分类器模型，以此来智能化检测警告的真假。警告分类器模型的训练需要大量真实世界的警告数据集，本文借鉴 [10] 中的差分分析法对开源项目多版本的警告打标签。警告分类器模型的输入需要警告的特征向量表示，本文参考 [11] 中的特征表示法提取警告代码特征向量，并使用 SoureMonitor[①]度量工具计算警告的手工特征，代码特征加手工特征即组成警告特征。本文主要围绕警告数据集的构建和警告特征的提取这两个点来优化警告分类器模型的分类效果。

1.2 国内外研究现状及分析

1.2.1 源码警告检测方法

源码静态扫描具有在不执行程序的环境下，只分析源代码以发现代码缺陷的能力。虽然它能够帮助开发者识别代码缺陷，但是往往会产生大量的警告，然而，这些警告里面还存在大量的误报。这是因为源码静态扫描工具为了提升扩展到大型和复杂软件系统的能力，使用了近似和假设技术，但是使检测结果变得不精确而导致大量误报 [7]。因此，开发者经常需要自己手动筛选错误警告以查找并解决真正的代码缺陷，一些开发者甚至杜绝使用源码静态扫描工具。

现有源码警告检测的方法越来越多，一些文献综述对这些方法进行了分类 [12]，主要分为聚类、排序和分类三种类别。基于聚类、排序、分类的源码警告检测技术是在警告源代码和警告报告的相关信息上，使用聚类、排序或者分

[①]https://www.derpaul.net/SourceMonitor/

类来辨别警告的真假。

1. 基于聚类的源码警告检测技术的基本解决思路是按照警告之间的依赖关系进行聚类。由于聚类后的警告以它们所依赖的报警器为主，不仅减少了需要检查的警告数量，而且消除了多余的检查工作 [13, 14]。Podelski 等人 [15] 为每个警告提出了一组基于语义的特征，并将具有相同特征值的警告进行分组。Le 和 Soffa[16] 通过收集有关警告相关特征的数据构建了一个相关图，该图可以整合不同路径和多个警告之间的警告相关性。Zhang 等人 [14] 提出了一种基于跟踪语义的警告关联算法来自动识别警告。Muske 等人 [17] 描述了一种通过重新定位来减少误报警告的新技术，该技术使用控制流的信息对相关警告进行分组。

2. 基于排序的源码警告检测技术的基本解决思路是对所有警告进行优先级排序，并使用手工特征来计算每个警告为真阳性的概率。Jung 等人 [18] 利用警告句法上下文作为贝叶斯的输入来计算每个警告为真阳性的概率，并根据真阳性的概率对警告进行排序。Kim 和 Ernst[19] 提出了一种基于从源代码存储库中挖掘软件更改历史特征的警告优先级算法。Williams 和 Hollingsworth[20] 提出了一种利用软件项目的源代码更改历史来驱动和帮助改进警告排序的方法。Kremenek 等人 [21] 解决了其他排序方法的弱点，即警告排名应该是自适应的。利用警告和用户反馈之间的相关行为进行警告排序。与第一类方法相比，这种方法需要检查所有排序的警告。

3. 基于分类的源码警告检测技术的基本解决思路是从警告中提取出一组特征，并在警告特征上构建一个警告分类器，以识别警告是真阳性还是假阳性。Ayewah 等人 [22] 讨论了生成的警告的种类，并将警告的类型分为误报、微小的缺陷和严重的缺陷。Ruthruff 等人 [23] 提出了一个基于从警告本身提取出来的 33 个特征训练得到的逻辑回归模型，以预测 FindBugs 发现的真实警告，并使用筛选方法丢弃预测能力低的特征, 以建立高性能的预测模型。Reynolds 等人 [24] 使用一组描述性属性来标准化误报的模式。一些研究已经使用了机器学习分类模型抽象出真实警告和虚假警告之间的差异，以自动识别警告的真假。Brun 和 Ernst[25] 提出了一种基于机器学习的方案，该方案基于程序属性构建模型，并使用构建的模型对可能导致潜在缺陷的程序属性进行分类。Heckman 和 Williams[26] 从不同警告特征集中筛选出 51 个候选特征集，并基于该候选特征集评估了 15 种机器学习算法，这是预测真实警告最全面的研究之一。此外，他们还提出了一个基准，用于评估和比较警告自动识别模型。Liang 等人 [27] 自

动构建训练集以计算不同特征的学习权重，然后通过使用这些学习权重计算警告的分数，根据警告的分数对警告进行排序和分类。Hanam 等人 [28] 提出了一种基于每个警告的上下文寻找相似代码模式的方法，用以区分真实警告和虚假警告。Flynn 等人 [29] 使用多种静态分析工具从警告中提取出 28 个特征，在 28 个特征上训练并测试了 4 种分类模型 [30]。

相较于前两种源码警告检测技术，基于分类的源码警告检测技术更为常见且研究底蕴更加深厚。本文构建的源码警告分类系统使用的正是基于分类的源码警告检测技术。系统遵循分类检测技术的实现思路，在警告数据集和警告特征上做了一些改进。系统自身构建了一个大型的真实警告数据集并提取了警告的融合特征，该警告特征不仅包含警告的语法结构，更是捕获警告的关键度量值，能够充分表示警告全面的特性。

1.2.2 警告数据集收集

由于程序表现出多样化和复杂的行为，警告分类的训练模型需要有错误和非错误代码示例的大型标记数据集。这对于 CNN、RNN、GNN 等高级神经网络模型尤其重要。警告代码较为常见的处理形式为方法体、字节码和程序切片这三种具体表现形式。

提取警告方法体是警告代码处理的最为简单一种方式，数据中只包含警告行所在的方法体（即“警告方法”）。但是，许多与警告相关的代码位置并不在其所在警告方法中。在很多情况下，警告产生的原因跨越多个方法和类。因此，这种警告数据表现形式并不是警告数据存储的完美方式。

警告字节码一般需要借助一些静态分析工具例如 Soot[①]来提取。字节码是源代码的一种表现形式，比较常见的有三地址码等。字节码因为其更贴近机器语言而受到静态分析研究者的欢迎。尽管警告字节码中包含了警告代码丰富的数据流、控制流信息，但是提取警告字节码都是基于警告类级别的，警告数据中引入了大量和警告无关的噪声。而字节码反编译成源代码比较困难，这也导致了字节码和源代码建立映射关系比较复杂，因此无法在字节码的层面上去除噪声，将警告字节码作为警告数据的表现形式也不是最好的方式。

警告程序切片提取技术是警告代码处理最为复杂的一种方式。Watson 开发了一个叫 Wala[②]的工具，能够计算出程序切片；Giffhorn[31] 基于 Wala 开发

[①]https://github.com/soot-oss/soot
[②]https://github.com/wala/WALA

出 Joana[①]工具，提供更多高层次的程序切片接口，基本满足使用者的多样化需求。程序切片是一种将程序缩减为最小形式的技术，给定程序中的一个固定点，通过删除不影响给定点行为的代码来完成切片计算，得到的切片在该固定点仍具有相同的行为。可以使用后向切片计算从警告行到程序入口点的程序切片，它涵盖了与警告行相关的所有代码位置。

Bissyand[8] 总结了现有的警告分类数据集存在以下局限性：

1. 方法级别的数据集，不提供解释警告如何发生的上下文信息。此外，该类型数据集通常不指定警告类型和位置。在许多情况下，函数级警告示例甚至不包括警告产生的根本原因。
2. 某些数据集（例如 CGD[32]）源自 NVD[②]或 CVE[③]中已确认的代码缺陷。尽管它们具有高质量的标签，但此类样本的数量有限，可能会导致模型训练过于拟合而使得学习结果毫无意义。
3. Juliet[④]和 S-babi[⑤]等合成数据集数据量很大 [33]。然而，它们是基于一些预定义的模式生成的，因此不能代表在现实世界程序中存在的各种行为。
4. 基于提交消息或代码差异的标记数据集。根据提交消息预测代码标签会产生低质量的标签。基于代码差异的方法假设错误修复提交中的所有函数都会引发代码缺陷，但在现实中可能并非如此。除此之外，这类方法难以识别警告的类型、位置。

1.2.3 警告特征表示方法

基于深度学习的方法已经广泛应用于学习警告源代码的特征表示上 [34]。Kamiya[35] 将警告代码转换为正则化的 token 序列，用于代码分类。SourcererCC[36] 通过利用 token 排序以及优化的倒排索引技术对 Kamiya 的方法进行了改进。除了词汇信息外，Deckard[37] 还用一些语法结构的信息丰富了警告代码特征，用于分类检测。Raychev 等人 [38] 采用 RNN 和 n-gram 模型进行代码补全。Wang 等人 [1] 通过深度神经网络从源代码中提取语义特征来做警告预测。DeepBugs[39] 通过 Word2Vec 检测基于名称的警告代码。White 等人 [40] 利用嵌入预训练的 token 递归自动编码器来捕获警告语法树的句法信息。

[①] https://github.com/joana-team/joana
[②] https://nvd.nist.gov/
[③] https://cve.mitre.org/index.html
[④] https://samate.nist.gov/SRD/testsuite.php
[⑤] http://arxiv.org/abs/1808.09897

TBCNN[41] 在警告语法树上使用自定义 CNN 来学习代码片段的向量表示。CDLH[42] 结合了 Tree-LSTM 来表示警告代码的功能语义。此外，Allamanis 等人 [43] 在程序图上执行门控图神经网络，该程序图跟踪相同变量和函数的依赖关系，以预测变量名称并检测变量的使用情况。DeepSim[44] 将代码控制流和数据流编码为语义矩阵，用于检测代码功能相似性。Tu 等人 [45] 将多种不同的代码表示，例如标识符、CFG 和字节码集成在一起训练学习向量表示。

Koc 等人 [7] 对当前警告表示方法作出了高层次的总结，主要可划分为四种学习方法：手工设计的特征（HEF）、词袋（BoW）、递归神经网络（RNN）和图神经网络（GNN）。上述警告特征表示研究思路基本都是在这四种学习方法的基础上衍生出来的。

1. 手工设计的特征。通过专家列出警告代码和报告的可测量属性来构建特征向量，每个属性都可以由一个或多个元素以数字方式表示，这些属性可以指示警告的真阳性或假阳性。Tripp 等人 [46] 确定了检测 JavaScript 程序的虚假跨站点脚本（XSS）警告报告的特征。警告特征涵盖了词法，定量和安全相关的属性。由此可见，手工设计的特征很需要特定方向的专业知识。一旦定义了特征表示，就可以使用大量的分类器和训练算法来学习如何进行预测。

2. 词袋。“词袋”（BoW）特征相较于警告源代码，提供了更加简单有效的表示 [47]。BoW 将警告源代码表示为源代码中所有的单词的多重集，忽略它们之间的顺序关系。每一个单词得到的特征向量的长度与字典中的单词数目一样多，特征向量中的每一单元表示源代码中是否存在特定的单词。BoW 有两种变体。第一个变体是检查单词的出现，这会产生一个二元特征向量表示。1 表示对应的单词在源代码中，0 表示不存在。第二个变体是计算单词出现的频率，这会产生一个整数特征向量，其中每个整数表示相应单词出现的次数。一般而言，“单词”指的是从程序切片中提取的 token。

3. 循环神经网络 RNN。在警告分类问题上，循环神经网络 [48]、[49]、[50] 已成为一种强大的特征表示方法，它将警告代码视为（任意长度）token 序列并自动学习序列中每个 token 的向量表示 [47]。RNN 从左到右处理具有任意长度 $x = \langle x_0, x_1 \ldots, x_t, \ldots, x_n \rangle$ 的单词序列。图 1–1展示了 RNN 的基本结构，其中 U 是输入层和隐藏层之间的权重矩阵，V 是隐藏层和输出层之间的权重矩阵。对于每个位置 t，RNN 的隐藏层值 h_t 不只是取决于当前的输入 x_t，还依赖于上一个隐藏层的值 h_{t-1}，将前一个隐藏层的值 h_{t-1} 作为 h_t 的权重矩阵 W 的权

重。h_t 的具体计算公式如下：

$$h_t = f(U \cdot x_t + W \cdot h_{t-1})$$

o_t 作为表示输出层值的向量，下式展示了其计算方式：

$$o_t = g(V \cdot h_t)$$

图 1-1中红色方框框出的部分，展示了一个标准的 RNN 单元。在多轮训练周期中，RNN 的网络参数会被联合评估优化。由于 RNN 具有独特的循环结构，其能够隐式地捕获 token 序列前缀 $x_0, \ldots, x_{t-1}$ 的相关上下文知识。与 HEF 或 BoW 不同，特征向量 o_t 直接针对分类任务进行了优化。这种优势是以可解释性为代价的，因此 o_t 的值对于人类来说比 BoW 或 HEF 更难解释。

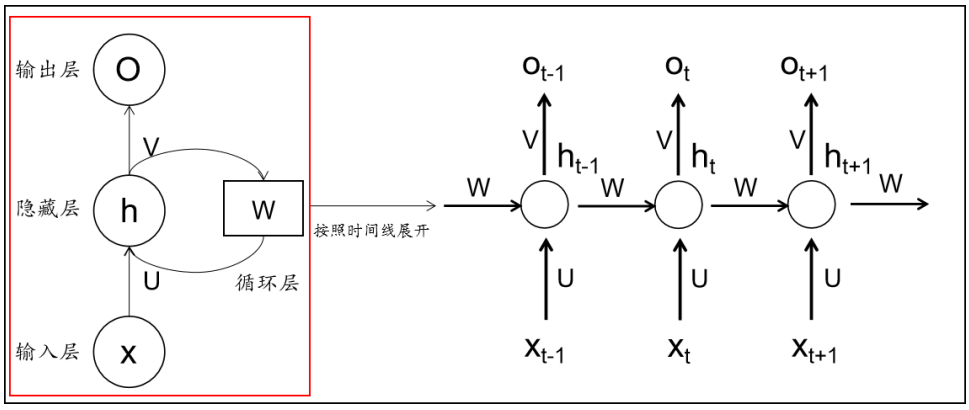


图 1-1: RNN 结构图

4. 图神经网络 GNN。由于程序具有更复杂的结构，可以更好地用图形表示。为了利用这种结构，因此产生了图神经网络，该网络使用来自相邻节点的信息 [51] [52] 来计算图中节点的向量表示。图的形式为 $G = \langle N, E \rangle$，其中 $N = n_0, n_1, \ldots, n_i$ 是节点集，$E = e_1, e_2, \ldots, e_j$ 是边集。每个节点 i 用向量 h_i 表示，该向量捕获了图上下文中节点的学习特征。边的格式为 $e = \langle type, source, dest \rangle$，其中 type 是边的类型，而 source 和 dest 分别是源节点和目标节点的 ID。向量 h_i 是迭代计算的，从时间 $t = 0$ 的任意值开始，并在每个时间步长 t 合并来自相邻节点 $NBR(n_i)$ 的信息，即 $h_i^{(t)} = f(n_i, h_{NBR(n_i)}^{(t-1)})$。函数 f 被定义为一个神经网络。

由此可见，目前的警告特征表示方法大多数都局限于软件度量、历史信息、源码本身其中一个方面做特征提取，而且每个方面提取的特征也不尽相同。由于现有方法并没有总结出一个全面的特征来表征警告，因此本文着重于

提取一种新的警告融合特征。

1.3 本文主要工作内容

针对上述背景，为了解决源码警告分类器模型在构建过程中出现的两大问题：（1）. 现有数据集多为合成测试数据集，不能代表真实世界的程序，因此缺少大型的真实警告数据集；（2）. 目前警告特征的提取要么局限于警告源码特征（部分方法仅仅将代码作为自然语言文本处理，无法获得警告的语法特征），要么只关注警告统计方面的特征，警告特征没有兼具其语法特征和度量特征。本文主要作出以下两大贡献：

1. 构建了一个大规模，真实项目的警告数据集。本文采用了一种基于差分分析的标记方法，用于构建一个大规模，真实项目的警告数据集。首先，选取 Apache 下 star 数目高、commit 次数频繁的项目，并下载近两年的所有版本；接着，结合三种警告比对策略（基于位置的比对策略，基于文本的比对策略，基于哈希的比对策略），版本间两两比对统计警告的出现情况；最后，采用差分分析法的标记策略对警告进行正误报标记。
2. 提取了一种全面的融合性代码特征。首先，使用程序切片、抽象语法树等静态分析技术对警告源代码做预处理；接着，构建基于抽象语法树的神经网络来学习源代码片段的向量表示，以捕获语句的词汇、语句级句法间的自然性。该模型将大的代码片段抽象语法树分解成小的语句语法树序列，通过递归编码多路语句树得到语句向量，利用神经网络模型学习代码语句向量的特征表示；然后，基于专家给出的意见，定义并提取了圈复杂度、警告危险等级、警告切片内函数调用数等 22 个手工定义特征；最后，使用扁平化技术融合代码特征向量和手工定义特征，并在警告特征上应用机器学习训练误报警告分类器模型。

1.4 本文的组织结构

本文的组织结构如下。第一章引言部分。本章概述了警告检测模型背景以及面临的问题，国内外研究现状和本文在解决该问题的主要思路。

第二章相关技术概述。本章讲述了系统中使用的警告比对策略，警告标记策略、程序切片技术、代码特征提取技术和警告分类器模型训练使用的技术。

第三章系统的需求与设计。本章首先对系统的功能需求、非功能需求以及对应的系统用户场景进行分析；其次，对系统使用到的技术栈和编码过程使用的专业性技术进行阐述，并结合 4+1 视图从多个维度对系统进行了剖析；最后，对系统进行了模块划分，并对系统的每个模块从架构设计、流程设计、核心类图三个角度切入进行了介绍、解析。

第四章系统的具体实现。本章主要是对系统的具体实现方式和逻辑进行了详细描述。按照系统模块划分，对系统中的警告数据集构建模块、代码预处理模块、代码特征提取模块、警告特征表示模块以及警告检测模型构建的顺序图和核心代码进行了阐述。最后对系统中的主要界面进行展示并讲解。

第五章系统测试与实验分析。本章主要是对系统的功能性需求和非功能性需求进行全面的测试。功能性需求的测试是针对第三章系统用例描述的功能进行逐个测试，验证系统提供的功能是否完备。非功能性需求的测试主要分为健壮性、性能测试和效果测试。健壮性和性能测试主要是对系统的稳健性进行测试。效果测试主要是验证了本文提出方法优于目前几种主流的做法。

第六章总结与展望。本章是对本文内容的总结以及对未来可优化方向的展望。对系统中源码警告分类器模型构建的整个流程做了一个总结，并列出了代码预处理和警告特征表示部分还可提升的地方。最后对系统中可以改进的三个点提出了自己的想法。

第二章　相关技术概述

2.1　警告标记策略

2.1.1　基于差分分析法的标记策略

基于差分分析法的标记策略是建立在项目的多个 commit 提交版本上进行标记工作的。首先，按照 commit 提交时间对所有提交的版本进行排序，以其中一个版本的警告作为数据集，后续版本的警告作为标记集；接着，应用警告比对策略，判定数据集中的警告在标记集中的出现次数；最后，根据警告在标记集中出现的次数标记警告的正误性 [53]。

可以将所有的警告分为三组：（1）警告在数据集中出现但是在标记集中消失，（2）在数据集和标记集中都出现的警告，以及（3）警告在数据中未出现但是在标记集中出现的。对于第（3）种情况，由于警告并不属于数据集，所以该情况的警告可以不予考虑。对于第（1）种情况，由于警告在后续版本中并未出现，可以认为该警告被后续提交修复，确实会引发代码缺陷，标记该类警告为正报。对于第（2）种情况，因为警告在后续版本中出现了，则证明该警告可能并不会引起代码缺陷，理应将其标记为误报。考虑到警告在标记集中出现的原因可能是静态分析器自身错误导致的误报，基于差分分析法的标记策略应用启发式方法处理该情况。同一警告可能出现在多个版本对中，如果在以后的版本中再次出现已修复的警告，更改此类警告的标签并将其标记为正报。只有当警告在标记集的所有版本中都出现，才将其标记为误报。

在基于差分分析法的标记策略中，最为关键的一环是判定数据集和标记集中的两个警告是否为相同警告。接下来的小节将会讲解在版本间是如何比对两个警告是否一致，即警告比对策略。

2.1.2　警告比对策略

针对给定的版本 S 产生一组警告，其中每个警告都由源位置 l 和警告类型 t 唯一标识。假设警告源位置由开始位置和结束位置给出，这些位置对引起警告

的代码段进行定界。因此，警告可以建模为三元组 (S,l,t)。警告匹配问题可以简明地表述如下：给定同一个项目的两个版本 S_p 和 S_c，其中 S_p 是 S_c 的父版本，以及两个警告三元组 $V_p = (S_p, l_p, t_p)$ 和 $V_c = (S_c, l_c, t_c)$，判定 V_p 和 V_c 是否表示相同的警告。

为了确定两个警告 (S_p, l_p, t_p) 和 (S_c, l_c, t_c) 是否匹配，有三种不同匹配策略的组合：基于位置的比对策略，考虑警告位置 l_p 和 l_c，基于片段的比对策略，考虑导致警告的程序文本，以及基于哈希的策略，考虑警告上下文的程序文本[10]。两个警告在它们属于相同类型时匹配，因此隐含的前提条件是 $t_p = t_c$。

基于位置的比对策略。基于位置的警告匹配的思想是使用 diffing 算法推导出从 S_p 中的源位置到 S_c 中的源位置的映射，然后在此映射下如果它们具有相同或几乎相同的起始位置，则匹配警告。然而，为 S_p 版本和 S_p 版本中的所有文件计算成对差异的成本太高了；因此，只计算两个版本中具有相同路径的文件成对差异。如果文件被重命名或移动到不同的目录，基于位置的警告匹配将无法匹配其中发生的任何警告。

当给定两个文件 F_p 和 F_c 时，其中 F_p 来自父版本而 F_c 来自子版本，差异算法本质上分别将 F_p 和 F_c 划分为行范围 $rp_1, \ldots, rp_n$ 和 $rc_1, \ldots, rc_n$。每对行范围 (rp_i, rc_i) 要么是匹配对，意味着范围在文本上是相同的；要么是差异对，意味着它们不相同。F_p 和 F_c 中的每一行都属于一个行范围。对于位置 l，$[l]$ 表示其起始行所属的匹配对范围，而 $\Delta(l)$ 表示从 l 到其匹配对范围起点的距离（以匹配对为单位）。

考虑位置 l_c 处的警告 V_c 和位置 l_p 处的警告 V_p，其中包含 l_c 和 l_p 的文件具有相同的路径。如果它们的范围一致，将它们匹配如下：（1）. 如果 $([l_c], [l_p])$ 是匹配对，那么警告是由在 S_p 和 S_c 之间没有变化的代码触发的，所以只在它们在区域内的位置完全相同时才匹配它们，即 $\Delta(l_c) = \Delta(l_p)$。（2）. 另一方面，如果 $([l_c], [l_p])$ 是一个差异对，允许位置稍微变化：如果 $|\Delta(l_c) - \Delta(l_p)|$ 某个阈值，则认为 V_c 和 V_p 匹配。

基于位置的匹配策略主要针对同名代码文件内容的修改，例如增删改等等，而代码块间的顺序性未发生变化的情况。基于位置的比对策略基本步骤：（1）. 使用差异算法对代码缺陷进行分片（相同文件），可以粗略地认为一个文件中的一个警告是一个代码块，一个文件可能会有多个警告，从而构成了多个分块。（2）. 使用匹配对算法判定 l_c 和 l_p 所在块是否匹配。匹配对算法：根据两个文件中的分块相对顺序，匹配 l_c 和 l_p 所在分块文本内容是否一

致。一致则表示该代码块在前后两个版本中未发生修改.（3）. 如果是匹配对，即可根据计算 Δ 的差值是否一致来判定是否是相同警告。Δ 不一致则认定不是同一个警告。如果是非匹配对，即代码可能发生修改，允许位置稍微变化。需要设定一个阈值。Δ 计算方式是警告行减去匹配对起始行。Δ 的差值小于等于阈值则视为匹配。

基于代码片段的比对策略。由于基于位置的匹配要求警告位置属于相应的行范围，因此这种匹配策略对于位置在版本之间发生了显著变化的警告将会匹配失败。使用基于代码片段的比对策略来捕捉在同一个文件中移动但由完全相同的代码片段触发的警告：如果 l_c 和 l_p 属于同一个文件并且 l_c 和 l_p 的源文本（即警告的开始和结束位置之间的文本）是相同的，即将它们匹配起来。

基于代码片段的匹配策略，主要针对同名文件中代码块间的顺序性发生了显著变化的情况。基于代码片段的比对策略基本步骤：l_c 和 l_p 所在区块的全内容匹配，即开始和结束位置之间的文本匹配。内容完全相同则匹配。

基于哈希的比对策略。基于位置的匹配和基于片段的匹配都不适用于在版本之间重命名或移动的文件中的警告。为了匹配这些警告，可以采用一种基于哈希的策略，尝试根据其周围代码的相似性来匹配警告。具体来说，对于警告 $V = (S, l, t)$，计算两个哈希值 $h < (V)$ 和 $h > (V)$：前者是从 n 个 token 计算到并包括 l 中的第一个 token，后者是从 l 中的第一个 token 开始的 n 个 token，其中 n 是一个固定的阈值（默认情况下，使用 $n = 100$）。如果 $h < (V_c) = h < (V_p)$ 或 $h > (V_c) = h > (V_p)$，则两个警告 V_c 和 V_p 被认为是匹配的。

如果从文件开头到位置 l 之间少于 n 个 token，则 $h < (V)$ 是没有意义的；类似地，$h > (V)$ 仅在位置 l 到文件末尾之间至少有 n 个 token 时才有意义。这避免了由于短 token 序列而导致的虚假匹配。

基于哈希的匹配策略，主要针对文件重命名的情况（非同名文件）。需要全局查找所有产生警告的文件，并使用上下文相似度匹配警告。相比于前两种比对策略，该策略复杂度最高。基于哈希的比对策略基本步骤：（1）. 计算 l_c 和 l_p 的 $h < (V)$ 和 $h > (V)$，前者是从 n 个 token 计算到并包括 l 中的第一个 token，后者是从 l 中的第一个 token 开始的 n 个 token 计算的，其中 n 是一个固定的阈值（默认情况下，使用 $n = 100$）。（2）. 如果 $h < (V_c) = h < (V_p)$ 或 $h > (V_c) = h > (V_p)$，则两个警告 V_c 和 V_p 被认为是匹配的。

在智能化源码警告分类系统中，警告标记策略被应用于构建警告数据集的模块中。系统结合三种警告比对策略的能力来判定一组警告是否相同，并统计

警告在提交版本中出现的次数，最后根据统计信息在版本间采用基于差分分析法的警告标记策略对警告打上标签。

2.2 程序切片技术

真实世界的程序很大，直接将警告所在类或者方法作为警告特征会引入代码噪声。因为代码中的许多部分与警告无关，可能会影响后续机器学习训练模型的效果。为了解决这个问题，Koc 等人 [4] 从报告的源代码行 [54] 计算待分析程序源码的后向切片，它包括所有可能影响警告行行为的语句 [7]。

2.2.1 程序切片计算原理

由于程序切片被定义为保留程序执行状态轨迹的某些投影，因此在讲解程序切片的计算原理前，首先需要介绍一下有向图、数据流图这两个数据结构；接着在数据流图的概念上描述变量集合和程序执行状态轨迹；最后，阐述是如何计算状态轨迹投影的。

有向图的结构是 $\langle N, E\rangle$，其中 N 是一组节点，E 是 NxN 中的一组边。如果 (n, m) 在 E 中，则 n 是 m 的直接前驱，m 是 n 的直接后继。从 n 到 m 长度为 k 的路径是节点列表 $p_0, p_1, \ldots, p_k$, 其中 $p_0 = n, p_k = m$，对于任意的 $i \in [1, k-1]$, (p_i, p_{i+1}) 在 E 中。

数据流图的结构是 $\langle N, E, n_0\rangle$，其中 $\langle N, E\rangle$ 是有向图。相较于有向图，数据流图的结构多出一个 n_0 节点。n_0 是数据流图的初始节点，因此存在从 n_0 到 N 中所有其他节点的路径。如果 m 和 n 是 N 中的两个节点，如果 m 在从 n_0 到 n 的每条路径上，则 m 支配 n。计算程序切片基本都是根据数据流图来做的。

在数据流图中，假定 V 是出现在程序 P 中所有变量名的集合，那么对于 P 中的每个语句 n（即 P 的数据流图中的节点），有以下两个集合，每个集合都是 V 的子集：$REF(n)$ 是在节点 n 处使用的一组变量，而 $DEF(n)$ 是在节点 n 处发生改变的一组变量。程序执行状态轨迹在每个语句运行之前需要对所有变量值进行副本保存，记载了程序运行时的每一个状态。

程序执行状态轨迹是计算程序切片过程中最为重要的一个数据结构。程序 P 中长度为 j 的状态轨迹是有序对的列表，可以将其表示为 $(n_1, s_1)(n_2, s_2), \ldots, (n_j, s_j)$。其中每个 n 在 N（P 中的节点集）中，每个 s 是记录在节点 n 上所有变量到具体值的映射。每个 (n, s) 在执行 n 之前会得到上一个

状态的 V 的值。需要注意的是，程序切片只处理最终能够停止的程序，因此特别排除无限状态轨迹的程序（例如死循环等等）。

计算程序 P 的切片接受的输入是一个二元组 (i, V)，其中 i 是需要进行切片的关注点语句，V 是语句 i 中涉及到的变量集合。切片准则 $C = (i, V)$ 确定投影函数 $projc$，切片准则只考虑 V 中所有变量的执行状态序列，并且只有当该语句是关注点 i 时，投影函数才会返回状态序列。因此，令 $T = (t_1, t_2, \ldots, t_n)$ 为执行状态轨迹，n 为 N 中的任意节点，s 为一个单射函数，记录所有变量到具体值的映射。则某一时刻的轨迹有：

$$proj'_{\langle i,V\rangle}((n,s)) = \begin{cases} \lambda & \text{n != i} \\ (n, s|V) & \text{n = i} \end{cases}$$

其中 $s|V$ 是指 V 中变量的单射函数，λ 是空字符串。现在将 $proj'$ 扩展到 i 的整个执行状态轨迹上，则有：

$$proj_{\langle i,V\rangle}(T) = proj'_{\langle i,V\rangle}(t_1) \ldots proj'_{\langle i,V\rangle}(t_n)$$

根据上述讲解，程序 P 在切片标准 $C = (i, V)$ 上的切片 S 是由一些程序的子集构成的，该子集保留其关注点 i 所有变量 V 行为的投影。因此，切片 S 具有以下两个属性：（1）.S 能够通过从 P 中删除 n 个语句（$n \geq 0$）获得。（2）. 保证 $proj_C(T) = proj_{C'(T')}$，其中 T' 是程序切片中的执行状态序列，$C' = \langle succ(i), V\rangle$，并且 $succ(i)$ 是原始程序中最靠近 i 的后继语句（如果 i 不在程序切片中）或者即 i 本身（如果 i 在程序切片中）。

2.2.2 程序切片的主要流程

使用 Joana[31]（一种 Java 程序分析框架）来计算后向切片。第一步是确定程序开始运行的入口点。对于待分析源代码，首先找到包含警告行的方法的调用层次结构，然后在这个层次结构中确定源代码中调用的方法作为入口点。如果程序是库，则此类方法可以是 API，也可以是主方法（这是默认入口点）或测试用例。接下来，计算程序依赖图（PDG），它由表示从入口点可到达的程序位置的 PDG 节点组成。然后，确定出现在警告源代码行中的 PDG 节点。最后，计算从那行到入口点的后向切片。

在智能化源码警告分类系统中，程序切片技术被应用于警告代码预处理模

块中。系统使用程序切片技术提取警告行的初始切片。由于初始警告切片可能会存在不可编译的情况，系统采用了一些静态分析技术完善初始切片结果。

2.3 代码特征提取技术

2.3.1 抽象语法树（AST）

抽象语法树（AST）是一种旨在表示源代码句法结构的数据结构。它已被编程语言和软件工程工具广泛使用。AST 的节点对应于源代码的结构或符号。一方面，与普通的源代码相比，AST 是抽象的，不包括标点和分隔符等所有细节。另一方面，AST 能够用来描述源代码的词法信息和句法结构。

构建 AST 的基本步骤。第一个步骤是词法分析。此过程涉及扫描源代码并将其分解为一组 token。在编程语言中，可以把 token 分为符号、保留字、字面值和标识符这四种类型。语法分析是构建语法树的第二个步骤，用于检查语法并根据输入的 token 序列构造一棵 AST。具体来说，在从词法分析中提取出一组 token 后，该步骤检查 token 的输入结构是否满足指定的语法。如果遵循语法，那么就会生成一棵 AST。

AST 基本结构。从语法规则构造的每棵 AST 都有一个根节点和 n 个子节点（$n \geq 0$）。语法规则中存在但没有重要意义的终结符号将被忽略并且不包含在 AST 中。如果语法规则的右侧仅包含终结符号，则根节点存储这些终结符号的信息。但是，如果存在至少一个非终结符，则 AST 的根节点以字符串的形式存储语法规则的标识，它的子节点表示语法规则的非终结符号 [55]。

2.3.2 控制流图（CFG）

控制流图（CFG）是一个有向图，图中每个节点都代表一个基本块，每条边代表块与块之间的控制流。基本块是由一系列连续语句组成的，其中控制流从开始语句进入并从结束语句离开，除了结束之外不会出现没有停止或者产生分支的情况。CFG 主要用于静态分析和编译器应用程序，它可以准确地表示程序单元内的流程 [55]。

CFG 构建方法。Cooper 和 Tim 等人 [56] 创建了一种算法来构建 CFG。该算法使用递归技术，将语句分解为更细的语句，直到它不能被分解。生成的 CFG 是将构成它的简短语句与一个特定规则组合的结果。不能分解的语句由一

个 CFG 节点表示，该节点具有相同的入口和出口块，即节点本身。

构建 CDG 的挑战。CFG 构建的挑战之一是处理子程序调用的情况，其中需要保存每个子程序的 CFG。在表达式中调用函数。每当构造语句的 CFG 时，都会有一个过程来检查是否存在对该语句中包含的任何表达式的函数调用。如果存在函数调用，则需要将表达式的 CFG 节点连接到被调用函数的 CFG 上。CFG 构造的另一个挑战是处理递归情况。必须保存由子程序构造的 CFG 的第一个节点。每次构建子程序的 CFG 时，首先检查该子程序的 CFG 之前是否已经构建过。如果之前已经构建过，调用它的表达式或语句将连接到之前保存的节点。如果从未构建过，则执行子程序中的 CFG 构建过程，并保存已构建 CFG 的第一个节点。

在智能化源码警告分类系统中，代码特征提取技术被应用于警告代码特征提取模块中。系统提取警告程序切片的 AST，并使用遍历算法将抽象语法树拆分为一组语句 AST。语句 AST 列表即为系统提取的警告特征表现形式。除此以外，抽象语法树由于其可编译的特性，还被应用在完善初始切片结果的算法中。CFG 作为另一种特征提取方式，被应用在对比试验中。

2.4 基于机器学习的警告分类模型

近年来，出现了多种分类技术，主要分为传统机器学习分类模型和基于神经网络的分类模型。我们简要概述了比较常见的几种分类器。

2.4.1 传统机器学习分类模型

k-Nearest Neighbors（KNN）。KNN 是一种基于监督学习的预测分类算法[57]。KNN 分类算法的基本思路是根据训练集中的 k 个最近邻数据来预测测试集中待测数据的分类结果。其中，k 是用户自定义的变量，对于文本分类问题而言，使用汉明距离度量并选择待测数据的最近邻。最终，将 k 个最近邻数据中出现次数最多的标签作为该待测数据的类别。KNN 十分简单，且易于使用。但是，在处理样本数量大或样本分布不平衡的数据集时，往往性能表现不佳。

朴素贝叶斯（NB）。朴素贝叶斯是一种应用统计学知识实现的分类器。正如其名，朴素贝叶斯分类器和贝叶斯定理是息息相关的。朴素贝叶斯分类器假定每个属性值对分类结果的影响与其他属性值无关，这在统计学上被称做条件独立性 [58]。NB 是一个条件概率模型：假设一个 k 分类的分类问题，用向量

$x = (x_1, \ldots, x_n)$ 表示 n 个特征的目标数据（自变量），可以使用如下公式计算 x 在分类 C_k 上的概率：

$$p(C_k|x) = \frac{p(C_k) \cdot p(x|C_k)}{p(x)}$$

朴素贝叶斯分类器可以用于文本，在数据样本大的数据集上训练具有较快的速度。由于它假定了属性之间的条件独立性，因此在属性之间存在关联的数据上训练分类器，往往得到的分类结果会很差。

随机森林（RF）。随机森林是一种由多棵决策树组成的分类器，其预测的分类结果是所有决策树的分类结果的众数。随机森林的实现算法如下：首先，从训练数据集中选择一个训练样本来构建分类器。其次，对于每个训练样本，生成一棵分类树，并且从测试数据中的几个随机样本中采取最佳分割。最后，通过聚合树的分类结果来预测新数据的标签。随机森林综合了多棵决策树的能力，避免了其过度拟合的缺点 [59]。它是最准确的学习算法之一。对于大部分数据集，它会生成高度准确的分类器，可以给出分类中哪些变量很重要的评估，并且减少了由于训练数据和测试数据之间的不平衡而导致性能不佳问题。

决策树（DT）。QUINLAN[60] 的决策树分类器是最著名的机器学习技术之一。决策树分类器是一种树状结构的分类器。在决策树中，每个决策节点对应于测试数据的某个属性的一个条件，并有多个分支，每个分支引导符合该条件的测试数据进入下一个决策节点。每个叶节点存储的是一种类别值，它是一个测试数据的决策结果。测试数据从根节点开始，递归地沿着当前节点进入符合其条件的子节点直至叶子节点，叶子节点的标签即为分类结果。决策树可用于对与训练数据具有相同特征的测试数据进行分类。由于决策树模型计算的成本相对较低，因此被广泛应用于统计、数据挖掘和机器学习等领域。使用决策树对测试数据进行预测时，模型给出分类结果的速度很快。但是决策树不适合处理缺失数据，可能会出现过拟合化的情况。

支持向量机（SVM）。支持向量机 SVM 是一种非概率型的分类器。给定一个训练子集，其中包含属于两个类别 c_1 和 c_2 的 p 维特征向量，通过找到最大边距 $p-1$ 维超平面用来划分两种类型。支持向量机首先根据每个类别的训练样本构建非线性分类模型。然后使用这些模型来预测测试集数据的类别 [61]。对于增量训练，SVM 会将训练数据映射到新的数据空间，并在新空间中找到下一个分离超平面。SVM 能够提高泛化性能 [62]，但是对于缺失数据十分敏感，缺失数据会影响分类器的分类效果。

2.4.2 基于神经网络的分类模型

卷积神经网络（CNN）。在深度学习中，卷积神经网络（CNN 或 ConvNet）是一种深度神经网络，其主要由输入层、卷积层、池化层和全连接层（线性结构）组成 [63]。一般来说，在卷积层和全连接层后还会加上一层激活层，使向量结果变得非线性化，避免分类结果的过拟合化。卷积层是 CNN 的核心层，其接受的输入是一个三维格式，分别代表数据体的宽度、深度和高度，因此 CNN 被广泛用于图像和视频识别的相关应用中。对于文本分类问题，使用 CNN 处理同样可以取得不错的效果。可以将深度设为固定值，模拟三维的数据形式作为卷积层的输入。卷积层的输出同样是三维的数据格式，因此需要使用池化层对向量做数据空间降维处理。最终使用全连接层计算分类概率值，并使用激活层让概率值非线性化，概率值最高的标签即为分类结果。相比于传统的神经网络，CNN 并非采用全连接的方式与前一层连接，因此避免了分类器模型过拟合的风险。

基于树的神经网络（TNN）。TNN 接受抽象语法树作为输入，通过自下而上的方式递归计算节点嵌入来学习其向量表示。最具代表性的基于树的模型是递归神经网络（RvNN）和基于树的 CNN（TBCNN）。

（1）. 递归神经网络（RvNN）是首先被提出用于处理自然语言和图像解析中的递归结构。给定一个树结构，假设一个父节点 y_1 有两个子节点 (c_1, c_2)，其中 c_1 和 c_2 是节点的词嵌入或中间向量表示。节点 y_1 的向量表示可以由下式计算：

$$p = f\left(W^{(1)}\left[c_1; c_2\right] + b^{(1)}\right)$$

其中 $W^{(1)}$ 是参数矩阵，f 是元素激活函数，$b^{(1)}$ 是偏置项。为了评估该向量表示的质量，解码层被用来重建子节点：

$$\left[c_1'; c_2'\right] = W^{(2)}p + b^{(2)}$$

然后通过 $E(\theta) = \|c_1 - c_1'\|_2^2 + \|c_2 - c_2'\|_2^2$ 来衡量训练损失。通过这种方式，RvNN 可以递归地计算和优化整棵树的参数，根节点的最终向量将代表给定的树的特征表示。White[40] 基于 RvNN 并结合了递归自动编码器来自动编码抽象语法树以检测代码克隆，其中抽象语法树由固定大小的输入被转换为完整的二叉树。

（2）. 基于树的 CNN（TBCNN）对树结构进行卷积计算，常常用于源代码分类等监督学习。它的核心模块是一个基于 AST 的卷积层，通过在整个 AST

上滑动来应用一组固定深度的特征检测器。可以由下式计算：

$$y = \tan\left(\sum_{i=1}^{n} W_{conv,i} \cdot x_i + b_{conv}\right)$$

其中 $x_1, \ldots, x_n$ 是每个滑动窗口内节点的向量，$W_{conv,i}$ 是参数矩阵，b_{conv} 是偏差。尽管原始抽象语法树中的节点可能有两个以上的子节点，但由于卷积的大小固定，TBCNN 将抽象语法树视为连续的完整二叉树。

相比于传统的机器学习分类模型，神经网络更受到研究者们的广泛使用。其一，神经网络在图像识别和自然语言处理中具有着天然的优势，它不需要使用者手动做一些特征工程，直接将数据输入进网络中，网络自动做特征工程并对参数进行优化。而传统的机器学习方法通常需要经过降维等一系列特征工程技术来提升模型的精确率。其二，神经网络可以学习大量的数据，数据量越大能带来更高的准确率。而传统的机器学习很容易达到准确率的瓶颈。

尽管传统的机器学习分类模型有着诸多缺点，一些研究者还是在上面做了一些实验。比如它的易于解释性，神经网络牺牲了不可解释性，从而获得高准确率，但是给参数优化带来了很大的挑战。并且它在小数据集上效果会更好且计算成本更低。所以，两种类型的分类模型都有被研究者在实验中使用到。

在智能化源码警告分类系统中，警告分类器模型主要被应用于警告分类器模型构建模块中。系统在警告特征上，训练警告分类器模型。除此之外，神经网络不仅可以作为分类器，还可以用来提取特征向量。系统使用循环神经网络对警告特征进行向量化表示，作为警告分类器模型的输入。

2.5 本章小结

本章主要介绍智能化源码警告分类系统在开发过程中使用到的相关技术。首先对构建警告数据集中使用的基于差分分析法的警告标记策略进行了详细描述，并对警告标记策略中运用的几种警告比对策略做了具体说明。其次对警告代码预处理中使用的程序切片技术进行了详细讲解，介绍了程序切片的计算原理和主要流程。接着，对代码特征提取技术进行了详细概述，包括抽象语法树（AST）和控制流图（CFG）。最后，对警告分类器模型的构建，包括传统机器学习分类模型和基于神经网络的分类模型进行结构概述，并分析了各个模型的使用场景和优缺点。

第三章　智能化源码警告分类系统的需求与设计

本章将要描述智能化源码警告分类系统的需求分析和设计。首先，本章对系统的目标和实现思路做了整体的概述；接着，本章对系统需求进行了全面的分析，包括功能性需求和非功能性需求，我们对此还设计了系统用例并作出详细的说明；然后，本章基于实现目标对系统进行总体架构设计和子模块的划分。最后，本章对每个子模块的实现流程做了进一步的详细阐述。

3.1　系统整体概述

由于源码静态扫描工具采用了过近似的技术，导致在源代码扫描中会出现大量的误报。因此，警告检测工作是很有必要的。综合目前警告分类器模型的研究现状，本文发现了两个有助于提升模型分类效果的点。其一，大规模的、真实世界的警告数据集有助于分类器更好地学习警告正误报之间的区别。其二，提取出一个全面的警告特征有利于更好地表现警告的特性。本系统针对警告检测的两大可优化点，采用差分分析法的标记策略构建警告数据集并融合警告的手工特征和代码特征得到全面的警告特征，从而构建出准确率更高的警告分类器模型，减少开发者的审计工作。

图 3-1是智能化源码警告分类系统的处理流程图。智能化源码警告分类系统可分为误报警告分类器模型构建和误报警告分类器模型应用两个部分。误报警告分类器模型构建：首先，筛选并下载 Apache 的 Java 项目，对已选项目源码做预处理并使用源码静态扫描工具扫描项目得到初始警告列表，结合三种警告比对策略（基于位置的比对策略，基于文本的比对策略，基于哈希的比对策略），采用差分分析法对警告进行正误报标记，警告数据集构建工作完成；其次，根据警告数据集所提供的报告信息，对警告代码做预处理。使用静态分析技术对源码切片处理以及抽象语法树提取并采取特定算法在源码层面上做代码补全和标识符抽象操作。接着，基于警告代码的抽象语法树，通过抽象语法树

拆分、Word2Vec 编码、训练特征一系列步骤，得到警告代码特征；然后，使用一些度量工具计算警告的手工特征（例如圈复杂度等等），并采用扁平化方式与代码特征融合得到警告特征；最后，在警告特征上训练警告分类器模型。使用警告分类器检测项目：首先，使用相同的源码静态扫描工具对待测 Java 项目进行扫描得到警告列表；接着，使用相同警告特征提取方式对警告做特征提取；然后，使用相同的警告表示方法对警告特征做向量化处理；最后，应用警告分类器对项目中的警告进行预测分类。通过上述步骤，可以有效地构建警告数据集并提取全面的警告特征，从而构建出效果更好的警告分类器，降低审计工作的人力成本。

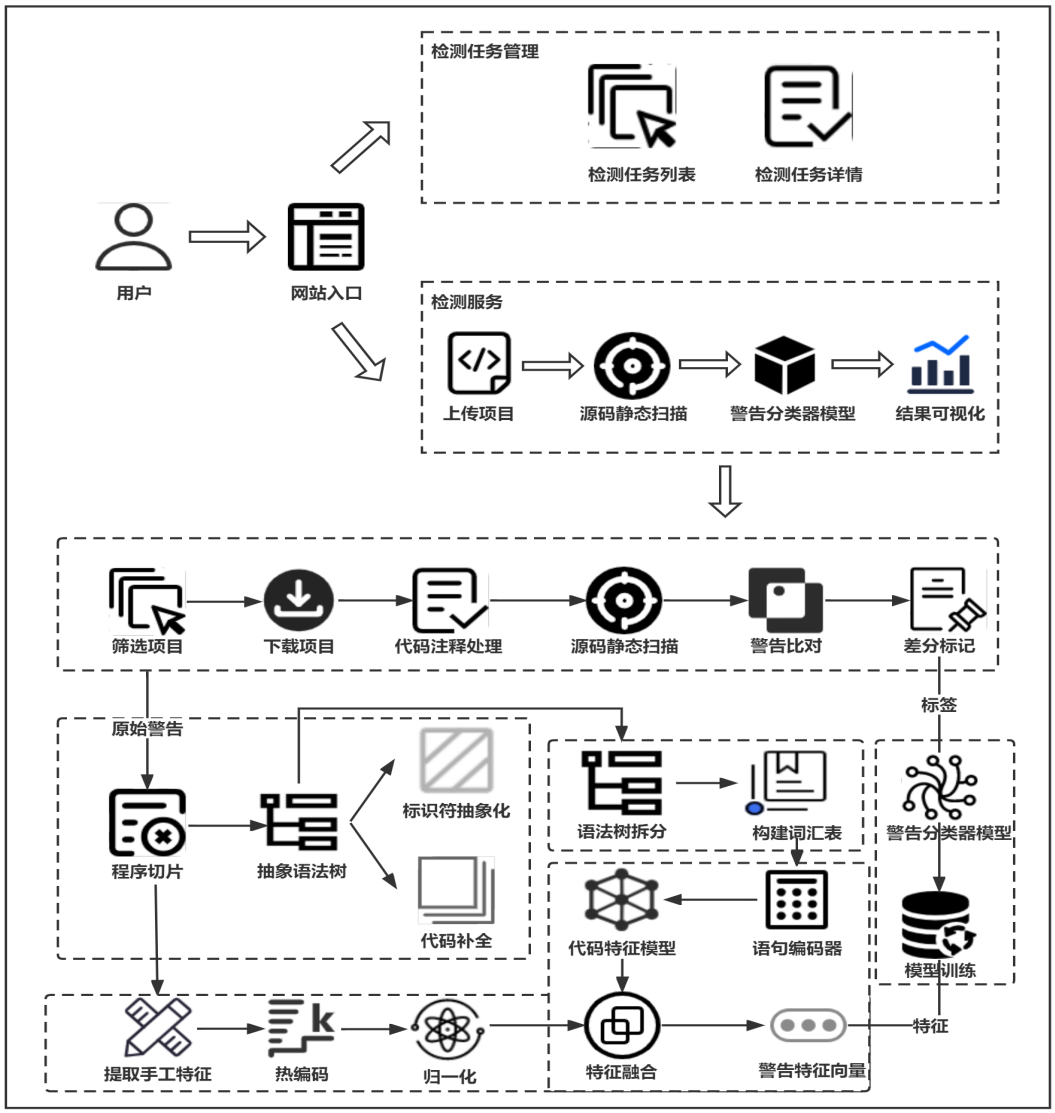


图 3-1: 智能化源码警告分类系统流程图

3.2 系统需求分析

3.2.1 功能性需求

智能化源码警告分类系统分为警告分类器模型构建和警告分类器模型应用两个部分。功能性需求按照这两个部分进行分析。主要需求如表3-1所示。

表 3-1: 功能需求列表

ID	需求名称	需求描述
R1	警告源代码静态扫描	用户上传待测 Java 项目，点击“开始扫描”按钮后，系统会先对待测 Java 项目做去除注释等预处理操作，接着对其进行扫描得到初始警告列表。
R2	查看检测任务列表	用户可以查看自己的所有警告检测任务，默认按照检测任务开始的时间进行排序，可以看到检测任务的基本信息，包括检测状态、检测时间、检测项目等。
R3	查看检测任务的初始警告列表	用户选定某个警告检测任务，可以查看源码静态扫描工具扫描获得的初始警告列表，每个警告包含警告类别、警告危险等级、警告代码信息等数据项。
R4	查看检测后的警告列表	系统完成某任务的警告检测后，即可查看检测后的警告列表，检测后的警告列表相比初始警告列表多了一个正误报标识的数据项，默认正报警告位列误报警告之前。
R5	训练警告分类器模型	在系统拥有误报检测的能力之前，需要训练误报警告分类器模型。系统存储了构建好的真实世界警告数据集，使用源码静态扫描工具扫描警告数据集得到初始警告列表。对警告列表采取代码预处理、警告特征提取、特征向量化表示一系列操作得到警告特征，在警告特征上训练警告分类器模型，获得检测警告的二分类模型，通过该模型可以得到检测后的警告列表。

在警告分类器模型构建流程里，首先，系统构建好大型的、真实世界的警告数据集；接着，系统提取数据集的代码特征和手工特征，并将二者融合得到向量化表示；最后，在警告特征上，训练警告分类器模型。

在警告分类器模型应用流程里，首先，系统需要从浏览器中获取用户上传

的待检测 Java 项目；其次，系统对待检测项目做去除注释等预处理工作并使用相同的源码静态扫描工具扫描预处理后的项目，得到初始警告列表；接着，根据警告提供的警告信息，经过警告代码预处理、警告代码特征提取、警告特征表示一系列步骤，获得警告向量化表示；最后，在警告向量上应用构建流程中训练好的模型，获得警告的分类值，即得到检测后的警告列表。

在检测任务管理页面，可以看到当前用户所有的检测任务详情，包括任务的检测进度和检测时间等。可以触发源码扫描和警告检测操作。

在警告列表管理页面，对于扫描完成的任务，可以查看初始警告列表并且触发警告检测。在警告检测结束后，可以查看检测后的警告列表。用户可以按照警告危险等级进行排序以及按照警告类型进行筛选，便于查看警告概况。

3.2.2 非功能性需求

系统的非功能性需求如表3-2所示。为了解决系统未来发生变动的问题，系统使用分层架构，便于添加新的功能，当有更加优秀的静态分析工具和能力更加强大的神经网络时，只需修改少量代码即可替换静态分析工具和模型训练使用的网络，增强系统的可扩展性。

表 3-2: 非功能需求列表

需求名称	需求描述
可扩展性	系统采用分层架构，各层之间解耦，便于添加新的功能。系统应该将使用源码静态扫描工具扫描和模型训练部分抽象出来，便于以后替换其他源码静态扫描工具和神经网络。
可维护性	在系统关键代码处需添加注释，尤其是在一些专业性程度比较高的步骤上，例如程序切片，抽象语法树以及模型构建等。在系统每个阶段性工作开始和结束时打印日志，便于维护人员发现 Bug。
可操作性	系统需提供良好的操作文档，前端设计应符合 HCI 设计规范。系统呈现给用户的界面应简洁明了，稍繁琐的操作需逐步拆解且提供简短的注意事项。除此以外，优化用户体验功能例如筛选按钮、搜索框等也需涉及。
健壮性	系统上线前，需构建覆盖率不低于 90% 的测试集并在测试集上进行测试，测试全部通过且在测试服务器上试运行 1-2 天无异常，该版本才可发布上线。

为了降低未来的维护难度，代码内添加注释和日志是不可或缺的，沉淀一些重要步骤的文档也是很有必要的。为了提升用户体验，系统需提供良好的操作文档，且界面需做到简洁明了，适当地给予用户一些指向性的反馈。为了避免因用户操作不当造成系统异常的隐患，系统需要考虑各种用户可能触发的操作，构建全面的测试集进行测试，并在测试机器上模拟用户试运行一段时间，挖掘尽可能多的潜在问题以确保系统的健壮性。

3.2.3 系统用例图

根据上述功能性需求的描述，系统用例图如图3-2所示。系统可划分为4个用例，分别为查看检测任务列表、触发警告扫描、触发警告检测、查看警告详细信息。

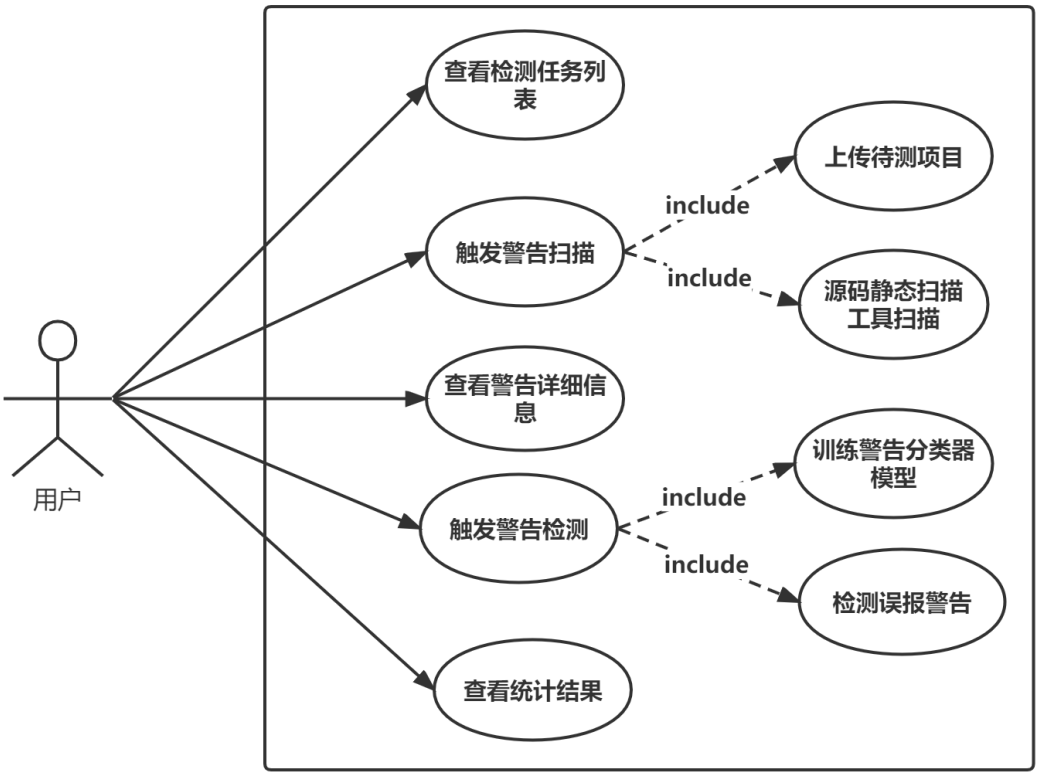


图 3-2: 系统用例图

3.2.4 系统用例描述

查看检测任务列表是系统的首页，系统用户可以在该页面上对所有检测任务的进度有一个总体的了解，便于用户发现并执行未开始扫描的检测任务，默认将未扫描的检测任务排列在前面，并在状态栏标记为灰色用来提醒用户。该用例描述如表3-3所示。

表 3-3: 查看检测任务列表用例描述

UC1	查看检测任务列表
参与者	系统用户，目的是查看检测任务的进度
触发条件	系统用户进入查看检测任务列表页面
前置条件	系统用户已经登录
后置条件	无
优先级	高
正常流程	1. 系统用户完成登录； 2. 系统显示该用户的所有检测任务，并将未开始的检测任务按照时间顺序前置显示； 3. 系统用户查看检测任务的状态； 4. 系统展示已完成、未开始、进行中的检测任务列表。
特殊需求	检测任务的不同状态应使用不同颜色的字号。

查看警告详细信息是本系统的基本功能。页面展示警告的详细信息，包括警告的报告信息和源码信息。用例描述如表3-4所示。

表 3-4: 查看警告详细信息用例描述

UC2	查看警告详细信息
参与者	系统用户，目的是查看警告的详细信息
触发条件	系统用户在警告列表中选择某个警告
前置条件	检测任务已经完成扫描
后置条件	无
优先级	高
正常流程	1. 系统用户在警告列表页面选择某个警告； 2. 系统显示该警告的统计信息。

触发警告扫描是本系统的一个自动化流程。虽然系统用户操作简单，只需点击选定检测任务的“开始扫描”按钮，但是在系统内部进行的处理往往耗时较长。触发警告扫描后，系统会调用内置的 SpotBugs 源码静态扫描工具，对检测任务中上传的 Java 项目进行扫描。扫描完成后得到初始警告列表，并持久化进数据库中。用例描述如表 3-5所示。

表 3-5: 触发警告扫描用例描述

UC3	触发警告扫描
参与者	系统用户，目的是触发系统警告扫描任务，启动警告自动化扫描。
触发条件	系统用户点击“开始扫描”
前置条件	系统用户已经创建好检测任务
后置条件	警告扫描完成，初始警告结果持久化进数据库中
优先级	高
正常流程	1. 系统用户在检测任务列表页面点击“开始扫描”； 2. 系统显示扫描任务的进度，扫描完成后可查看初始警告列表； 3. 系统用户点击查看报告按钮，即可显示警告列表。

表 3-6: 触发警告检测用例描述

UC4	触发警告检测
参与者	系统用户，目的是触发误报警告模型的使用。
触发条件	系统用户选择“警告检测”按钮
前置条件	选择检测任务已经完成扫描
后置条件	初始警告列表经过检测发生变化
优先级	高
正常流程	1. 系统用户在检测任务列表页面选择“警告检测”； 2. 系统显示警告检测的进度，并在检测完成后，可查看警告检测的结果； 3. 系统用户选择查看检测结果，即可显示检测后的警告列表。

触发警告检测是本系统最核心的功能。在检测任务完成扫描获得初始警告后，系统方能允许用户执行警告检测任务。当系统用户选择某个检测任务，并

点击“开始检测”按钮，系统内置的警告检测模型即开始对初始警告列表做检测。由于耗时较长，系统会人性化地在前端展示检测的进度。用例描述如表3-6所示。

3.2.5 数据库设计

经过分析，本系统的数据可以分为三类：第一类是用户的身份信息，主要用来帮助实现用户的登录验证。该类数据需要持久到数据库中，为此设立 User 数据表。使用该系统前必须得先注册账号，获得账号后方可登录成功进入系统的主页面。

第二类是能可视化的数据，主要用于系统界面展示。用户在使用警告检测服务前需要创建检测任务，因此自然需要创建检测任务 Task 表。创建好检测任务后，用户可以对检测任务发起扫描、检测等操作，系统响应执行操作产生警告列表，并将结果返回给用户。因此，需要构建警告 Alarm 表。

第三类是无需可视化的数据，主要是系统运行过程中的一些中间数据，包括程序切片、警告特征等。系统将该类型数据保存为 pkl 格式的文件，置于服务器上。该类型的数据采用 Dataframe 的方式存取，在这里就不赘述了。

与用户身份信息相关的数据表实体关系图如图3-3所示，其中只有一张表，即用户 User 表，字段如表3-7所示。id 作为用户的一个唯一标识。username 用户名，password 用户密码和 email 用户邮箱都需要用户手动输入，其中 username 和 password 是必填项，email 是可选项。

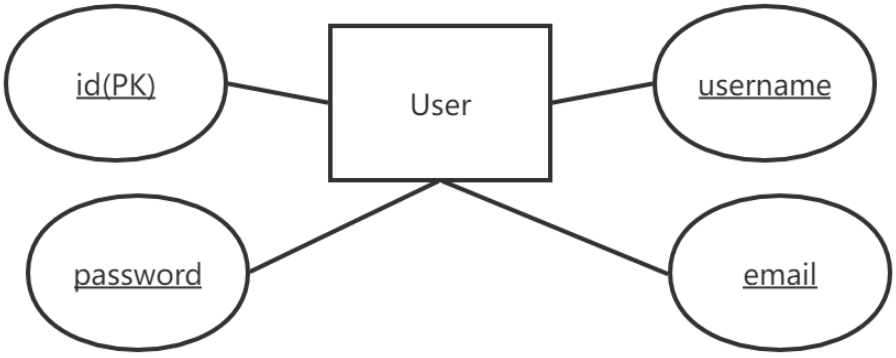


图 3-3: 用户身份信息相关表的实体关系图

表 3-7: User 表

字段名	数据类型	属性	说明
id	VARCHAR(255)	PK	主键，User 的唯一标识
username	VARCHAR(64)	NN,UNIQUE	表示用户名
password	VARCHAR(64)	NN	表示用户密码
email	VARCHAR(64)	N	表示用户邮箱

与警告相关的数据表实体关系图如图3-4所示。其中检测任务表 Task 保存了一个检测任务的详细信息，警告表 Alarm 记录了一个警告的具体信息。一个检测任务经过系统扫描、检测可以获得一组警告。因此，检测任务和警告之间存在一对多的关系。警告通过外键 taskid 可以得知属于哪个检测任务。

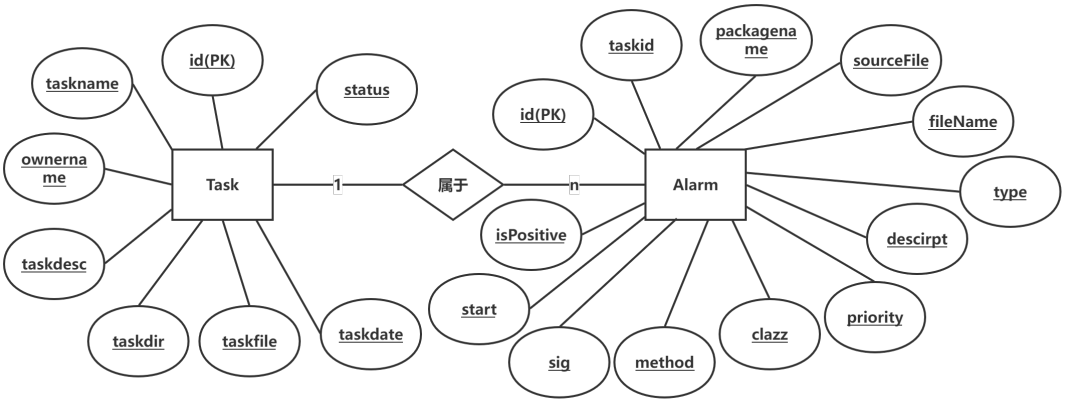


图 3-4: 警告相关表的实体关系图

表3-8是对检测任务的字段说明。其中，id 作为 Task 的唯一标识。ownername 表示任务所有者。taskname 表示任务名，用户在界面需手动填写。taskdesc 表示任务描述，用户可选填。taskdir 表示任务工作目录。taskfile 表示任务检测项目，用户在界面需手动上传检测项目。taskdate 表示任务创建时间，由系统自动生成。status 表示任务状态，有“已创建”、“已扫描”、“扫描中”、“已检测”和“检测中“五种状态。

表 3-8: Task 表

字段名	数据类型	属性	说明
id	VARCHAR(255)	PK	主键，Task 的唯一标识
ownername	VARCHAR(64)	NN	表示任务所有者
taskname	VARCHAR(64)	NN	表示任务名
taskdesc	VARCHAR(64)	N	表示任务描述
taskdir	VARCHAR(255)	NN	表示任务工作目录
taskfile	VARCHAR(255)	NN	表示任务检测项目
taskdate	VARCHAR(64)	NN	表示任务创建时间
status	VARCHAR(64)	N	表示任务状态
id	VARCHAR(255)	PK	主键，Alarm 的唯一标识

表 3-9: Alarm 表

字段名	数据类型	属性	说明
taskid	VARCHAR(255)	FK,NN	外键，表示该警告出现在此任务中
packageName	VARCHAR(64)	NN	表示警告所属包名
absolutePath	VARCHAR(200)	NN	表示警告所属文件的绝对路径
sourceFile	VARCHAR(200)	NN	表示警告所属文件的相对路径
fileName	VARCHAR(64)	NN	表示警告所属文件名
type	VARCHAR(64)	NN	表示警告类型
descript	VARCHAR(128)	NN	表示警告描述
priority	VARCHAR(64)	NN	表示警告危险等级
clazz	VARCHAR(128)	NN	表示警告所属类名全称
method	VARCHAR(64)	NN	表示警告所属函数名
sig	VARCHAR(200)	NN	表示警告所属函数签名
start	INT	NN	表示警告起始行
end	INT	NN	表示警告末行
isPositive	BOOLEAN	N	表示警告是否为正报

表3-9是对警告的字段描述。其中，id 作为 Alarm 的唯一标识。taskid 作为外键，记录警告属于哪一个检测任务。packageName 记录警告所属包名。

absolutePath 记录警告所属文件的绝对路径。sourceFile 记录警告所属文件的相对路径。fileName 记录警告所属文件名。type 记录警告类型。descript 记录警告描述。priority 记录警告危险等级。clazz 记录警告所属类名全称。method 记录警告所属函数名。sig 记录警告所属函数签名。start 记录警告起始行。end 记录警告末行。isPositive 记录警告是否为正报。

3.3 系统总体设计

3.3.1 系统整体架构设计

智能化源码警告分类系统的架构如图所示 3-5，接下来我们将按照模块间的联系从交互层面做出详细的阐述。

系统前端使用 Vue 框架，Vue 作为一套渐进式开发框架，适合自底向上逐层构建。因为其只关注视图层，开发非常易于上手且与第三方类库结合的比较方便。系统使用的组件库是 Element，作为目前 Vue 生态圈最热门的 UI 组件库，元素简单轻便，非常符合国人的观赏体验。前后端交互采用了 Ajax 异步请求，大大提升了用户体验。js 代码库使用的是 jquery，兼容大部分浏览器且符合服务端人员的开发习惯。

系统服务端采用了分布式架构解决方案，通过将特定的用户服务独立出来部署，实现客户端和服务端的分离，提升系统的可维护性。用户服务包含获得用户上传待测项目、扫描状态查看、检测结果查看、任务管理查看等。系统服务则包含整个误报警告智能检测构建模块。

数据服务部分提供存储服务，主要存储警告数据集、警告特征。这些数据都会被持久化到 MYSQL 数据库中。随着用户不断地使用，每一次的检测任务以及检测过程中产生的一些数据（警告列表、警告特征、检测结果等）也会被存储在数据库中。

3.3.2 系统模块划分

如图 3-5所示，系统可拆分为五个子模块。第一个模块是警告数据集构建，主要负责构建一个大型的真实世界警告数据集。第二个模块是代码预处理，主要负责对警告源码做切片处理并对提取的切片进行代码补全和标识符抽象操作。第三个模块是代码特征提取，主要负责从警告切片代码中提取出包含

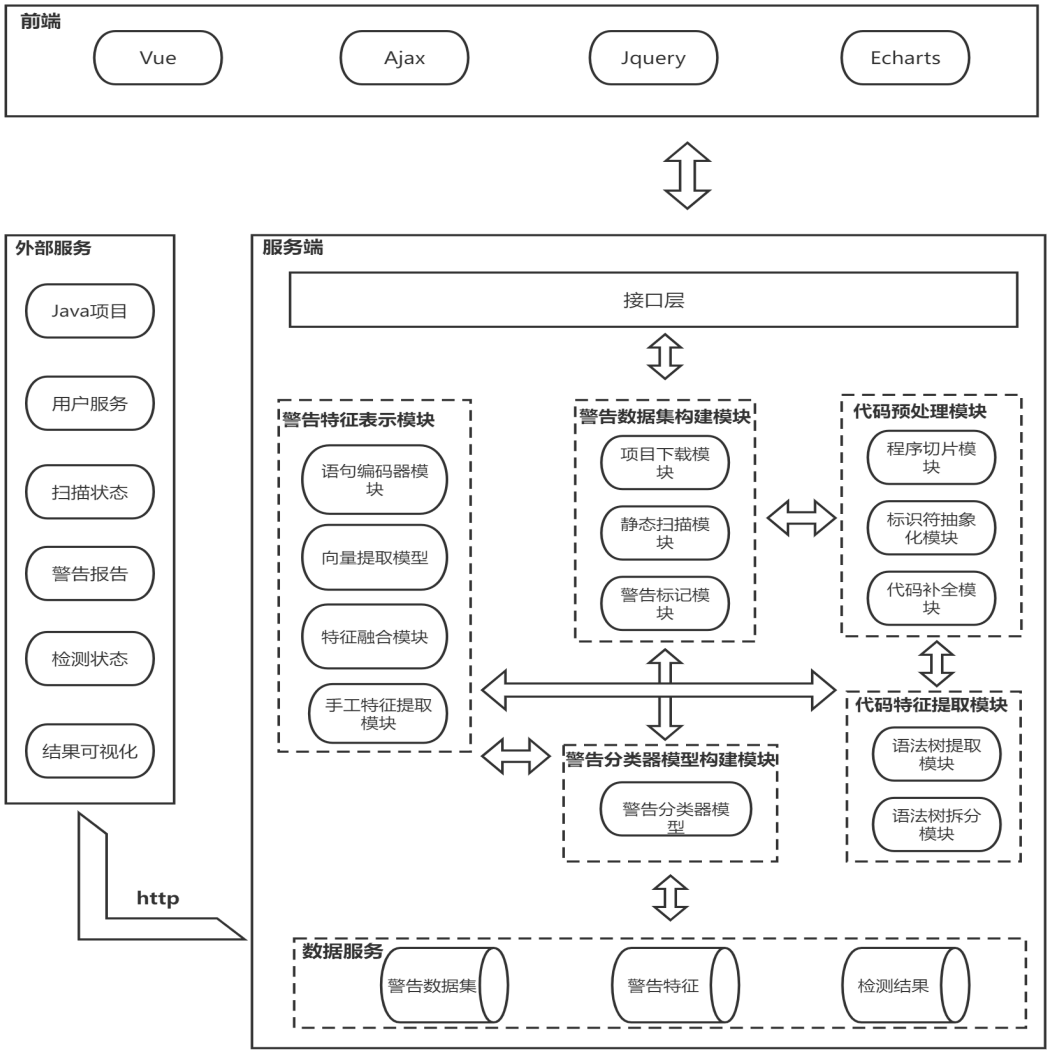


图 3-5: 智能化源码警告分类系统架构图

代码句法结构的代码特征。第四个模块是警告特征表示，主要负责将警告特征包括代码特征和手工特征，转化成数字化向量。第五个模块是警告分类器模型构建，主要负责在警告特征上学习警告分类器。接下来根据警告分类器模型构建的流程来介绍每个模块的设计。

警告数据集构建模块。首先，系统从 maven 中央仓库中挑选 apache 下 star 数目高、更新频繁的项目；然后，下载已挑选项目近两年的所有版本并使用源码静态扫描工具 SpotBugs 扫描获得原始警告；最后，系统结合三种警告比对策略（基于位置的比对策略，基于文本的比对策略，基于哈希的比对策略），采用差分分析法对警告进行高精确的正误报标记。

代码预处理模块。首先，系统对警告列表中的警告项做程序切片处理，切片的入口点为警告项所在函数，sink 点为警告项；接着，系统提取警告源代码的抽象语法树，根据上一步得到的程序切片行号结果删除无关语句；最后，系统基于上一步修正的抽象语法树，根据语法树节点类型做标识符抽象化处理。

代码特征提取模块。首先，系统提取预处理后的警告切片抽象语法树，在语句级别的粒度上将语法树划分成多棵语句语法树，即每棵警告抽象语法树都由 n 棵语句语法树组成（n 为警告程序切片的语句数）；接着系统遍历每棵警告抽象语法树，获得抽象语法树的 token 序列；然后根据 token 序列训练警告的语料库；最后，系统遍历每棵语句语法树，将节点 token 与语料库中的索引建立映射匹配。使用列表的方式嵌套保存映射结果，保留语法树节点之间的层次结构。

警告特征表示模块。首先，系统构建高效的基于抽象语法树的神经网络来对语句语法树进行编码，并学习编码获得的语句向量以得到整个警告代码的向量表示；接着，系统使用度量工具 SourceMonitor 计算警告的手工特征；最后，系统使用扁平化技术将代码特征与度量特征加以融合。

警告分类器模型构建模块。首先，系统从警告数据集中提取警告标签并和警告特征做一一映射，作为警告分类器模型训练的特征标签输入。接着，系统使用深度学习框架搭建警告分类器模型；最后，多轮模型训练直至模型收敛，警告分类器模型构建完成。

3.3.3 4+1 视图

本部分从物理视图、开发视图、场景视图、进程视图和逻辑视图这五个角度来阐述系统的总体设计。其中场景视图是从用户需求的角度来，由于在前面的系统用例描述部分已经展开描述，在这里就不赘述了，具体如图3-2所示。

物理视图如图 3-6所示。物理视图是从系统的部署以及各组件之间的通信作为切入点展开对系统的描述。客户端经过防火墙向服务端发起 http 请求建立连接。客户端向服务端发起 http 请求，即可将检测任务分发给 App Server。App Server 分为 preParеData 和 false_detect，其中 preParеData 是负责数据集构建以及对警告代码做预处理的服务，false_detect 是负责误报智能化检测的服务。在系统使用过程中生成的警告初始报告、警告数据集、警告特征以及检测任务都会被持久化进数据库或者置于服务器中。

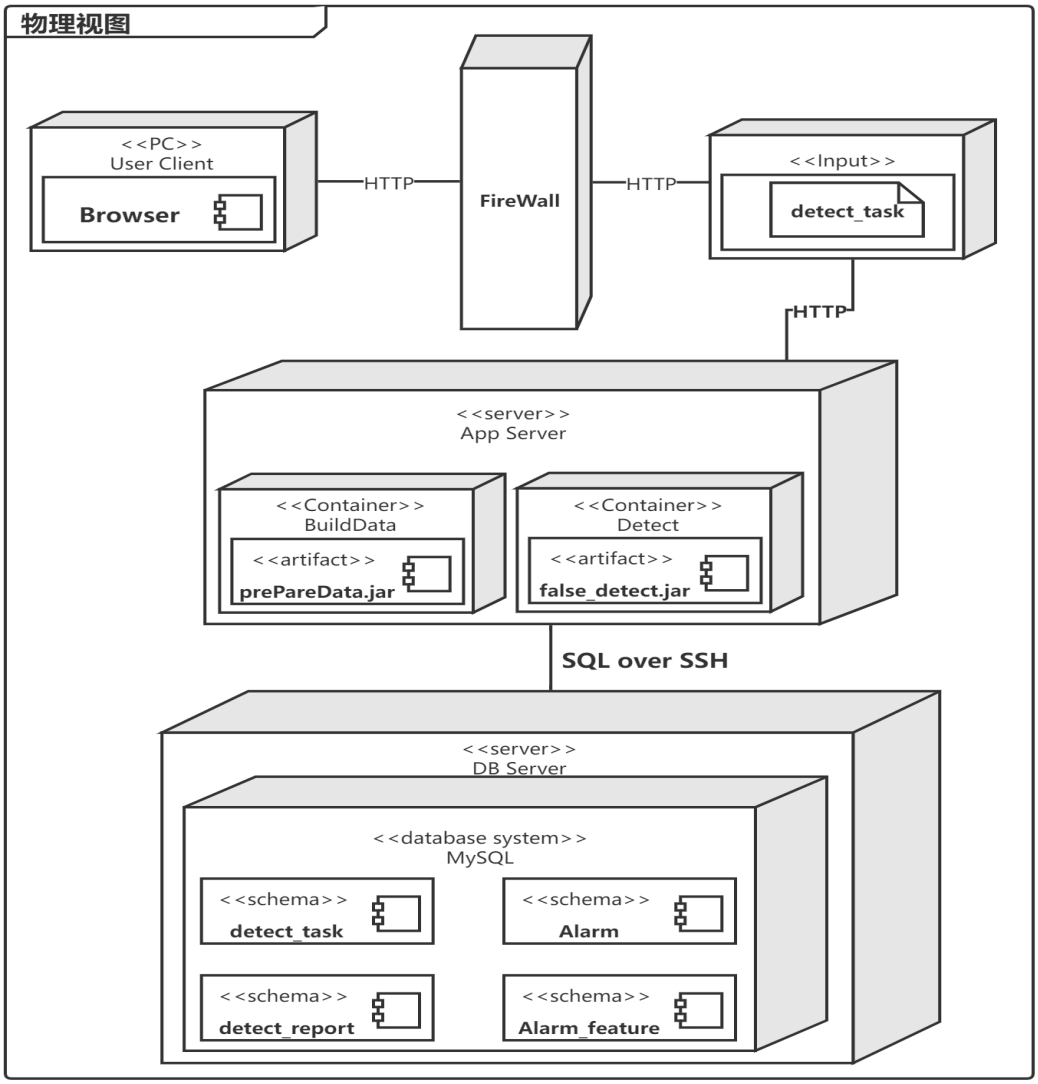


图 3-6: 物理视图

开发视图如图 3-7所示。开发视图是从开发者角度。UI 部分包含系统前端所使用的技术框架以及需要展示的一些核心要素。主要是 Vue、JS、Templates 以及程序切片和警告列表的展示。Service Logic 是服务端部分，按照分层架构的方式划分系统的层级。最上层是 Controller 层，负责和客户端直接交互，包括 TaskController 负责检测任务管理，ScanController 负责源码静态扫描，DisplayController 负责统计结果可视化展示，DetectController 负责源码警告检测。Controller 层负责接收用户请求，具体实现委托 Service 层处理。Service 层是服务真正的实现层，实现了 Controller 层提供的对外服务接口。Service 层会调用系统中的多个子模块来实现接口定义的功能。Scan 进行 Java 源码静态扫

描，ExtractFeature 提供特征提取服务，CodeAnalyzer 提供提取程序切片和 AST 服务，Detect 提供警告分类器模型构建和检测服务。Tech Service 是系统所依赖的已有技术，包括数据库 MySQL、机器学习框架 Pytorch、程序切片工具 Joana、AST 工具 JavaParser 和 Javalang，这些技术都被应用在我们的系统中。

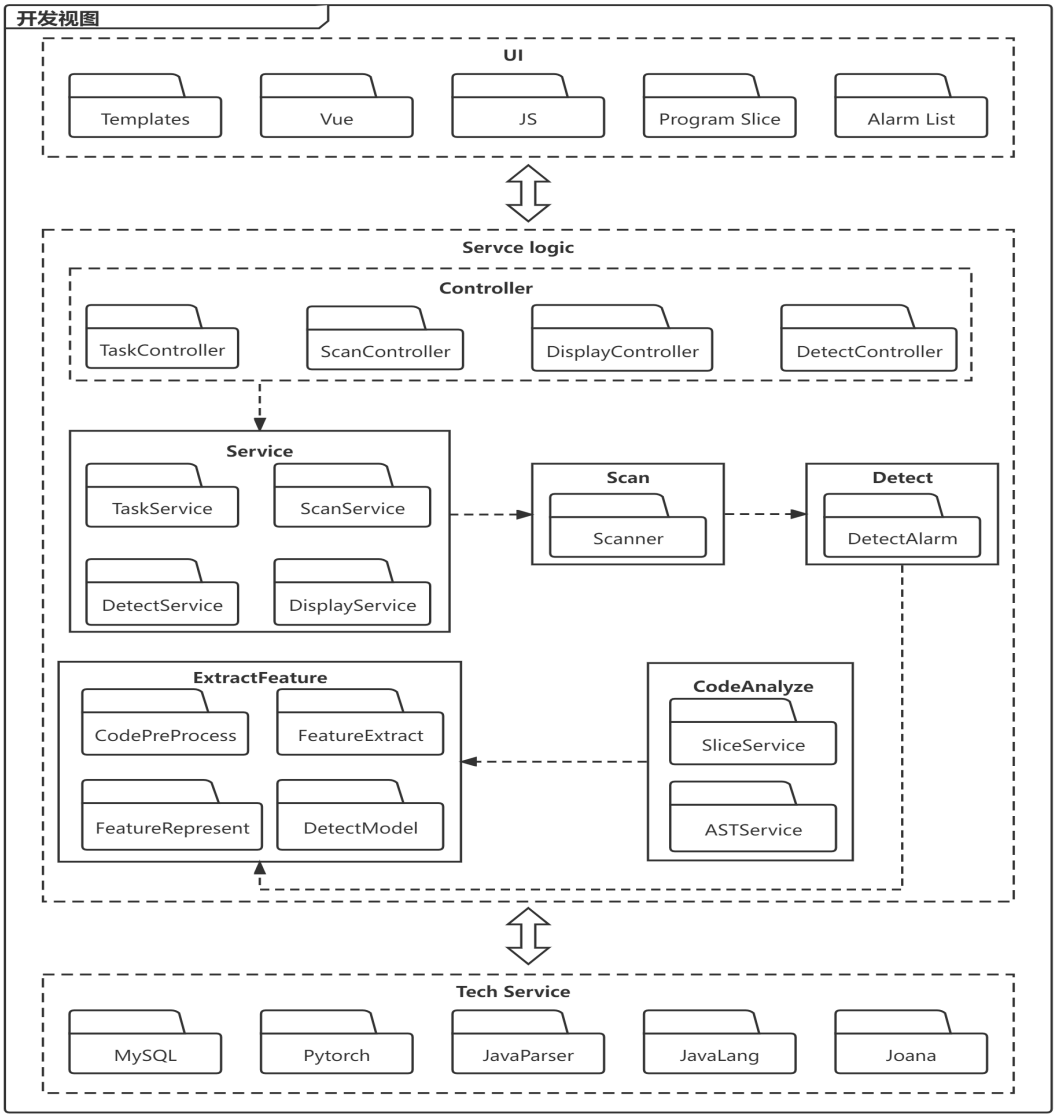


图 3-7: 开发视图

进程视图如图 3-8所示。进程视图是从用户。系统由主线程开始启动，当进行静态扫描时，异步调用静态扫描线程，扫描结束后将扫描得到的警告列表持久化到数据库中。当进行警告检测时，系统异步调用警告检测线程，主线程采用轮询的方式向警告线程发送心跳，获取检测状态，检测结束时返回检测结

果并存入数据库中。当进行结果展示时，系统同步调用结果展示线程，各项指标计算得出后，返回给主线程显示。

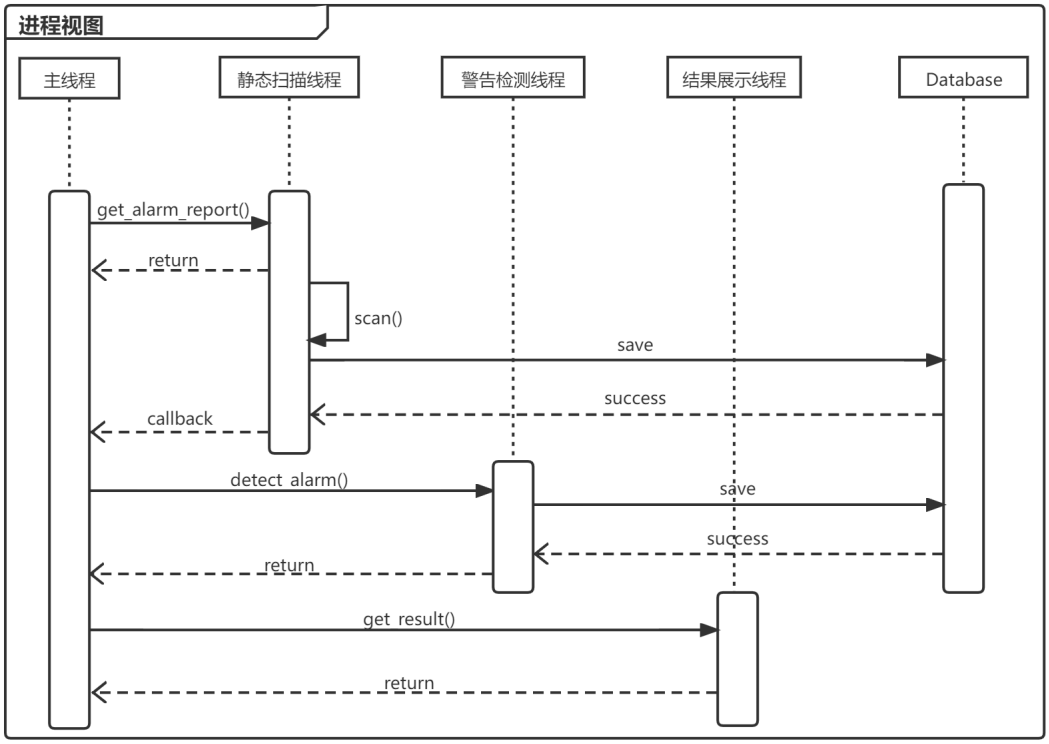


图 3-8: 进程视图

逻辑视图如图 3-9所示。逻辑视图是从运行流程角度对整个系统做子模块划分以及模块间关系的分析。AlarmScan 负责对待测项目做源码静态扫描，分为 PreProcessService 和 AlarmScanService。PreProcessSerivce 负责对源代码进行去除注释预处理操作，AlarmScanService 负责使用源码静态扫描工具对预处理过的项目进行扫描。源码静态扫描结束后，需要对扫描得到的警告列表进行检测。做误报警告检测首先需要有一个精度较高的警告分类器模型，DetectModel 即为事先训练好的警告分类器模型，分为 FeatureExtraction 和 ModelTraining。其中 FeatureExtraction 负责提取警告特征，ModelTraining 负责基于提取的特征上进行警告分类器模型的训练。有了警告分类器模型后，就可以对扫描得到的警告列表进行检测。AlarmDetect 负责警告列表的检测工作，分为 FeatureExtraction 和 FeatureFusion。FeatureExtration 负责提取警告特征，警告特征类别又可分为 ASTFeature 和 MetricFeature，其中 ASTFeature 是从警告代码 AST 中抽象出来的特征，MetricFeature 则是警告代码的度量特征。

FeatureFusion 负责对上一步提出的 ASTFeature 和 MetricFeature 做特征融合。TaskManage 主要负责检测任务的查看和管理功能，记录检测任务的唯一标识、检测用户等，委托 TaskManageService 具体实现。ResultShow 主要负责检测结果的展示，委托 ResultShowService 具体实现。

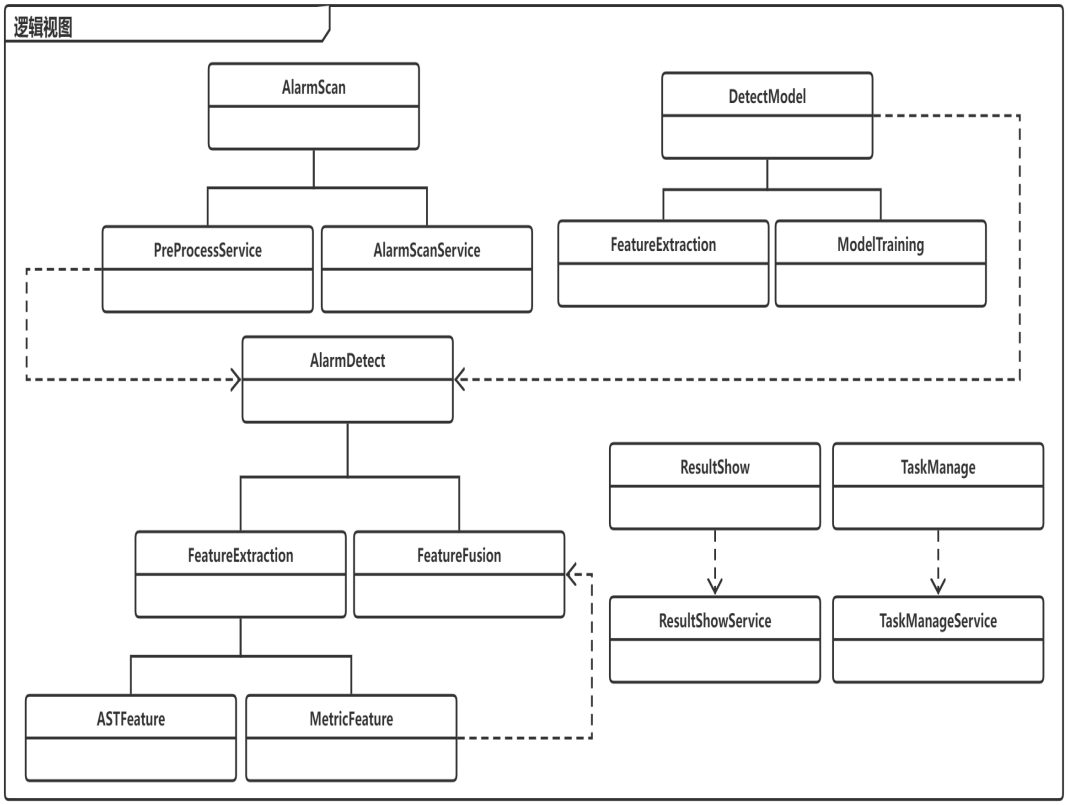


图 3-9: 逻辑视图

3.4 警告数据集构建模块设计

3.4.1 架构设计

如图 3-10所示, 警告数据集构建模块的主要目的是在项目多版本报告的基础上标记警告，构建真实世界的警告数据集。首先，筛选 Aphache 旗下 star 高，commit 次数多的项目，并下载近两年项目所有的 commit 版本。这样，对于每一个项目都对应多个版本。使用源代码静态扫描工具 SpotBugs 分别对每个版本扫描获得初始警告列表。接着，在每个项目的初始警告列表上，结合警告

比对策略，应用警告标记策略标记警告，将标记好的警告数据集持久化入数据库中。

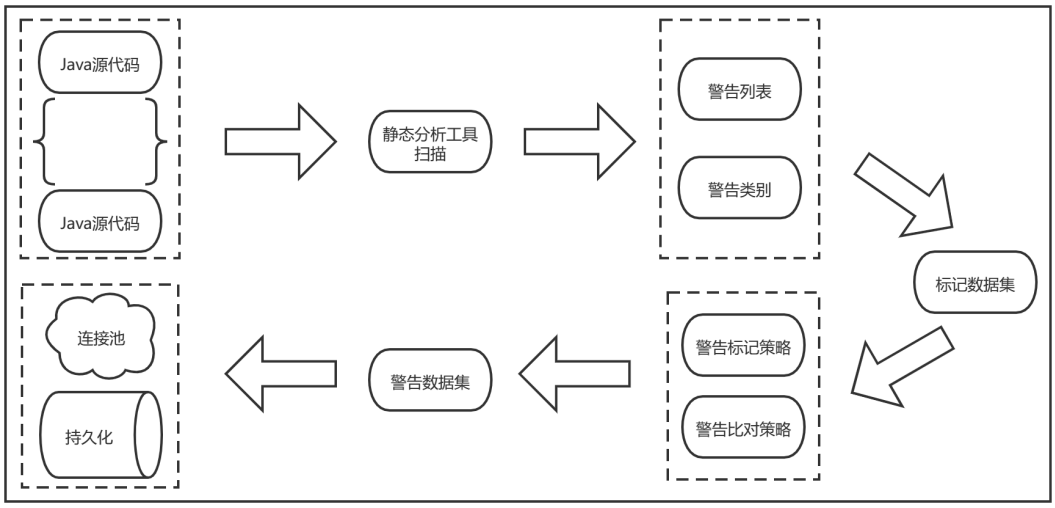


图 3-10: 警告数据集构建架构图

3.4.2 流程设计

警告数据集构建模块流程如图 3-11所示。该模块主要根据系统需要大量数据集的需求，构建真实世界的警告数据集。具体流程如下：

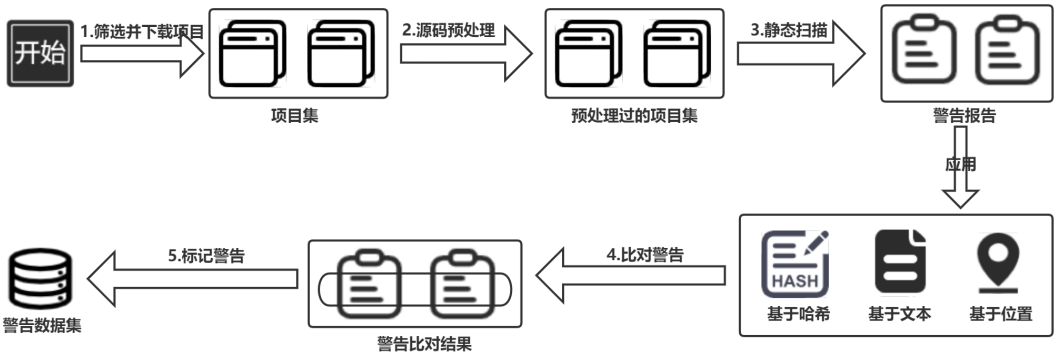


图 3-11: 警告数据集构建流程图

筛选并下载项目。本系统在 2019—2020 周期内根据项目的 commit 次数、star 个数对项目排序筛选，选择 commit 频繁，star 较高，并且代码规模尽可能大的项目进行下载，以获得数量更多的警告。

源代码预处理。本系统通过静态分析技术提取源代码抽象语法树并删除语法树中的所有注释节点，使用保持源码格式的输出方式将修改后的语法树转化成源代码形式。这样，在警告标记过程中，可以去除注释对标记结果的影响。

静态扫描。本系统使用源代码静态扫描工具 SpotBugs 对项目的每个版本进行警告扫描，获取项目的初始警告列表。为了获得更多的警告及警告危险等级，本系统将 SpotBugs 的扫描等级设置为 low，这样得到的警告列表包含 low、medium、high 三种等级的警告。

比对警告。本系统使用三种警告比对策略，在待标记警告的后续版本中统计相同警告出现次数，作为差分分析法标记步骤的输入。相较于单一的警告比对策略，使用三种警告比对策略比对警告，能够有效地标记出更多的误报警告。

使用差分分析法标记警告。本系统应用三种警告比对策略，使用差分分析法标记警告，生成带标签的警告列表，即为警告数据集。警告数据集需持久化进数据库中，后续需要对警告信息做进一步处理。

3.4.3 核心类图

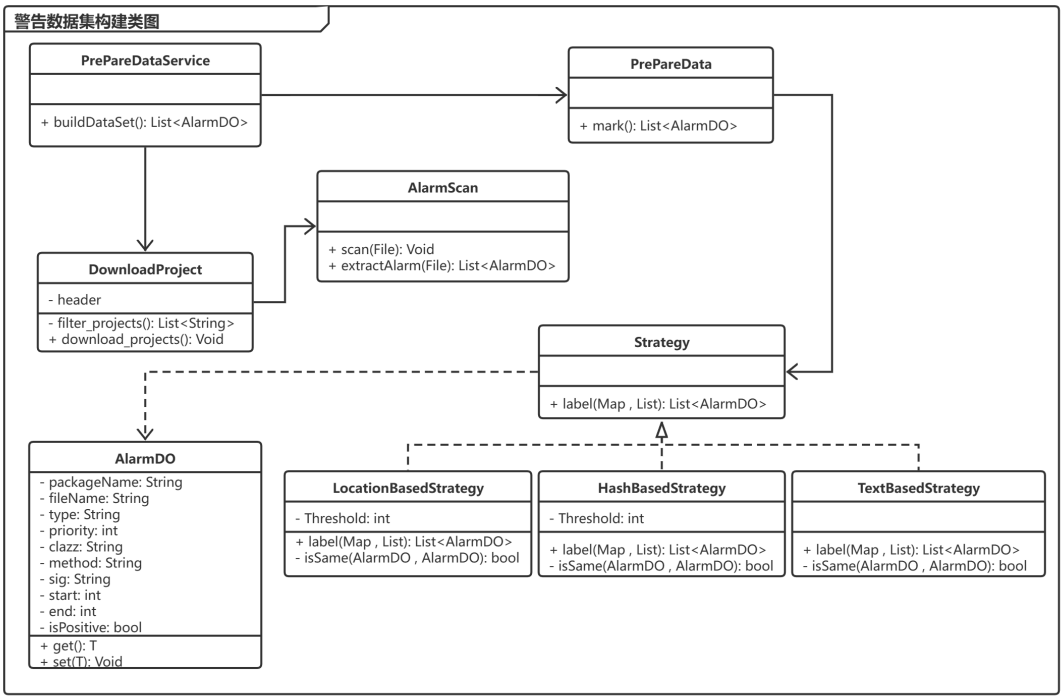


图 3-12: 警告数据集构建类图

如图3-12所示是警告数据集构建模块的核心类图。PrePareDataService 是对外提供数据集构建服务的类，提供构建数据集服务（buildDataSet）。构建数据集分为项目收集，警告扫描和项目标记三个部分，PrePareDataService 分别委托 DownloadProject 类，AlramScan 和 prePareData 类做这三部分工作。

实际的项目收集依赖 DownloadProject 类。DownloadProjects 类主要包含 filter_projects 和 download_projects 两个方法。filter_projects 负责过滤 maven 中央仓库中的项目集，以挑选出符合要求的项目。download_projects 负责根据已挑选项目的仓库地址，自动化下载项目代码。

系统的警告扫描依赖 AlarmScan 类。AlarmScan 类主要包含 scan 和 extractAlarm 两个方法。scan 负责使用静态扫描工具扫描项目获得警告报告。extractAlarm 负责从警告报告中提取警告信息，并将其封装成 AlarmDO 类。

项目标记阶段依赖 prePareData 类。prePareData 类主要包含 mark 方法。mark 方法负责对原始警告列表打上标签。针对不同的警告比对策略，系统使用了策略模式。基于位置的比对策略（LocationBasedStrategy）、基于文本的比对策略（TextBasedStrategy）、基于哈希的比对策略（HashBasedStrategy）具体实现了比对策略（Startegy）接口，封装不同比对策略的处理逻辑。

3.5 代码预处理模块设计

3.5.1 架构设计

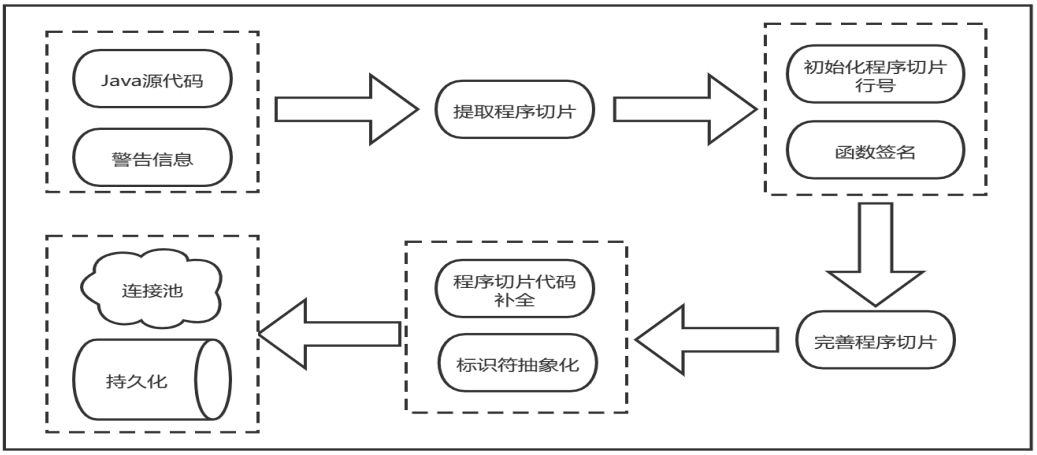


图 3-13: 代码预处理架构图

如图 3-13所示，代码预处理模块的主要目的是将警告所在行处理成更能表示其特征的代码形式。代码预处理，首先应用静态分析工具 joana 对源代码做切片处理，将源代码缩减为自身的一个子集，以隔离与警告不相关的代码位置；接着使用静态分析技术将切片结果可编译化并去除无关的代码噪音；最终该模块得到的输出需持久化到数据库中。

3.5.2 流程设计

代码预处理模块流程如图 3-14所示。代码预处理模块的处理流程主要是使用程序分析和抽象语法树等静态分析技术，并结合一些合理的抽象化规则来对警告代码做预处理工作。下面是代码预处理模块的主要步骤：

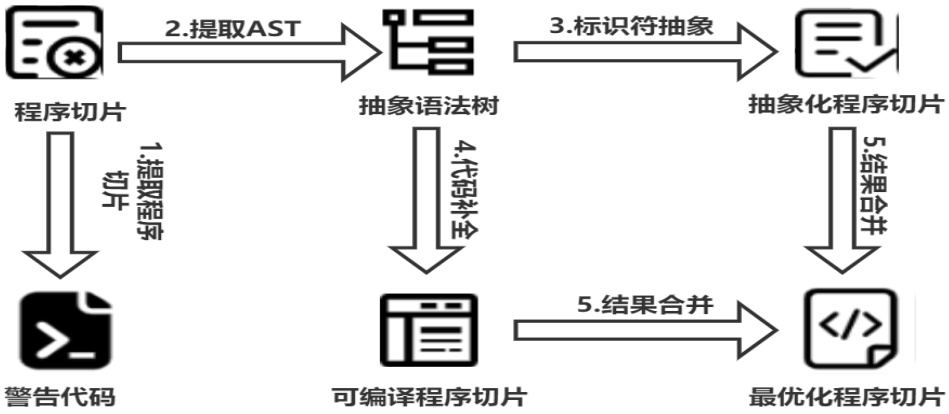


图 3-14: 代码预处理流程图

程序切片提取。在这项工作中，是为了捕获每个警告的上下文。系统提取了警告代码中所有影响和受相应报告语句影响的语句。从警告的语句开始，我们使用 Joana 在程序中进行后向的过程间切片，从而得到程序切片的初始行号。通过这种方法，不仅能去除程序中大量不相关的语句，而且通过整个程序中的控制/数据依赖关系精确捕获警告上下文。然而，需要留意的是，使用 Joana 工具获得的程序切片往往不具备可编译型。

提取抽象语法树。提取抽象语法树提取的是警告行所属函数的语法树结构，目的是为后续标识符抽象和代码补全步骤提供输入。使用 javaparser 根据警告所在行定位所属函数的抽象语法树，获得方法体的初始语法树结构。

标识符抽象。程序往往包含大量的标识符，它们的命名约束和风格也是多

种多样的。这可能会导致模型难以捕捉警告的一般模式。此外，模型会简单地学习某些项目中标识符的特征，以及简单地将特定标识符与相应的警告标签映射。为了避免这个问题，本系统在提取特征之前抽象了所有标识符。提取的切片中的变量、函数名和常量被替换为常见的符号名。本部分从以下三个方向做抽象化处理。（1）. 抽象数字和字符串文字：使用抽象值替换出现在切片中的数字和字符串文字。两位数字用 N2 代替，三位数字用 N3 代替，四位或更多数字用 N4+ 代替。负数和科学记数法中的数字应用类似的转换。系统提取字符串文字列表，并用 STR 后加一个唯一数字替换字符串。（2）. 抽象程序特定的词：从数据集中抽象出出现次数少于特定次数或仅出现在单个程序中的某些单词，将它们替换为 UNK。（3）. 从标识符中提取英文单词：许多标识符是由多个英文单词组成的。例如，Java 标准库中的 getFilePath 方法由三个英文单词组成：get、File 和 Path。为了使模型更具泛化性并减少词汇量，将任何 camelCase 或 snake_case 标识符拆分为它们的组成词。这三种抽象通过减少给定数据集的词汇量和泛化性来提高学习效率，帮助系统训练更通用的模型。

程序切片代码补全。程序切片代码补全的目的是补全代码缺失行，保证程序切片的可编译性。Joana 可以提取代码的程序切片，但是提取出来的初始切片通常不足以达到可编译的高标准。因此，需要借助抽象语法树结构，根据初始程序切片的行号，删去不相关的语法树节点，保留函数中维持可编译所需的最少语句，获得完整的程序切片。

结果合并。结果合并的目的是为了将标识符抽象和代码补全处理后得到的结果合并在一起。在代码补全后的程序切片上做标识符抽象处理，从而完善程序切片，使其兼具泛化性和可编译性的特质。

3.5.3 核心类图

如图3-15所示是代码预处理模块的核心类图。CodePreProcessService 是对外提供代码预处理的类，支持程序切片提取（doSlice）和函数方法体提取（sliceFuncBody）这两个功能。CodePreProcessService 通过 JoanaSlicer 类，SliceHandler 类和 ReplaceVals 类之间的协作完成代码预处理流程。

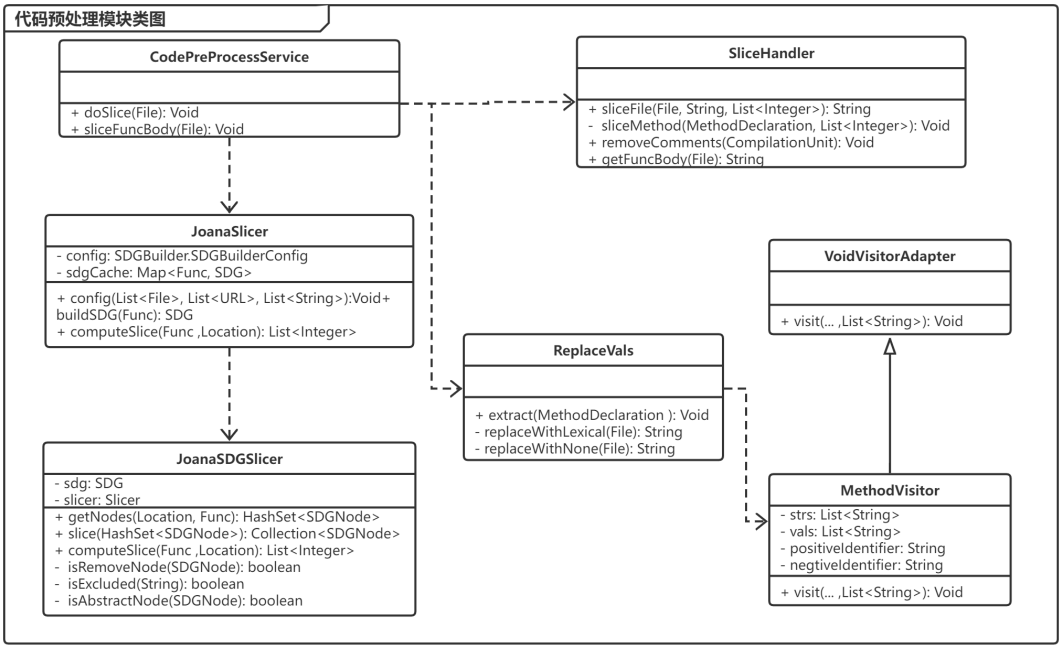


图 3-15: 代码预处理类图

程序切片提取依赖 JoanaSlicer 类。JoanaSlicer 类主要包含 config，buildSDG 和 computeSlice 三个方法。config 负责配置程序切片的属性，包括解析范围，解析对象等等。buildSDG 负责依据配置项和警告信息寻找函数入口点并构建系统依赖图 sdg。computeSlice 负责借助系统依赖图 sdg 计算警告的程序切片，获得警告切片的初始行号，具体逻辑委托 JoanaSDGSlicer 实现。JoanaSDGSlicer 类需要通过 JoanaSlicer 类生成的系统依赖图初始化并内置了一个程序切片计算器（Slicer）。JoanaSDGSlicer 类包含获取切片点（getNodes），判断是否删除节点（isRemoveNode，isExcluded 和 isAbstractNode），计算程序切片（slice 和 computeSlice）的功能。其中 computeSlice 调用了 slice 函数，并删除了无用节点后得到初始程序切片行号。

代码补全步骤依赖 SliceHandler 类。SliceHandler 类主要包含 sliceFile，sliceMethod，removeComments 和 getFuncBody 四个方法。sliceFile 负责对警告进行代码补全操作，sliceMethod 作为私有函数供 sliceFile 方法调用，以对警告所在方法体做增删改 AST 节点处理。removeComments 负责去除警告代码中的注释。getFuncBody 负责从抽象语法树 AST 中提取方法体并以字符串的形式返回给 CodePreProcessService 类。

标识符抽象阶段依赖 ReplaceVals 类。ReplaceVals 类主要包含 extract，

replaceWithLexcial 和 replaceWithNone 三个方法。extract 函数是标识符抽象的函数入口，调用 replaceWithLexcial 或者 replaceWithNone 进行具体的处理操作。replaceWithLexcial 和 replaceWithNone 函数区别在于前者保留了源代码的格式，而后者对源代码格式做了美化（本系统使用的是 replaceWithNone 函数，由于对比实验需要，故实现了 replaceWithLexcial）。MethodVisitor 是标识符抽象最核心的类，继承于 VoidVisitorAdapter。

3.6 代码特征提取模块设计

3.6.1 架构设计

如图 3-16所示, 代码特征提取模块的主要目的是从警告程序切片代码中提取出可以表示其语法特征的特征形式。

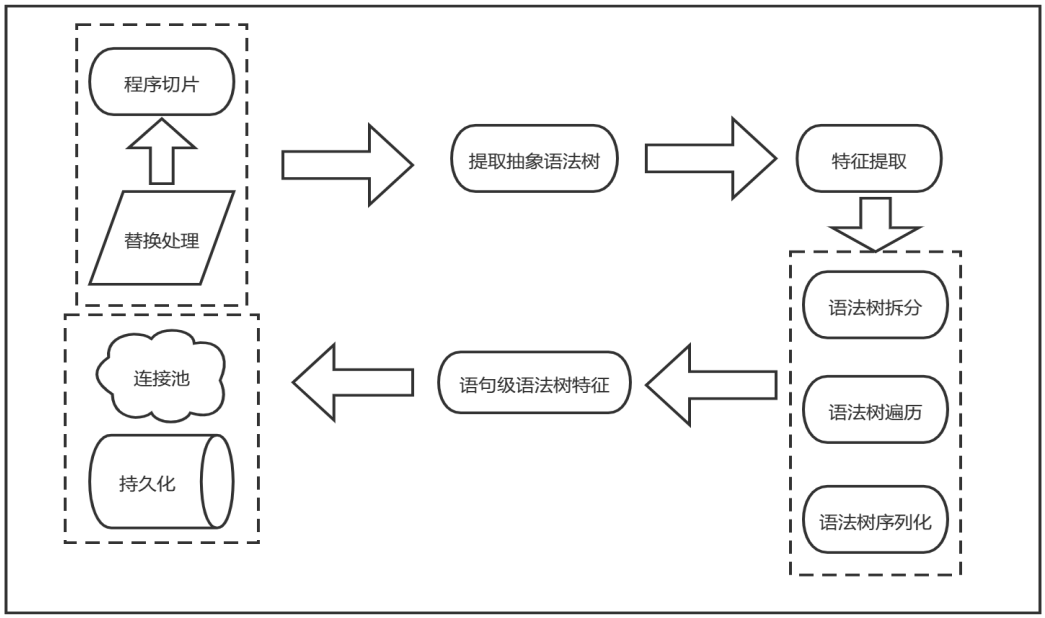


图 3-16: 代码特征提取架构图

首先，拆分语法树并构建语句树序列。系统通过现有的语法分析工具将源代码片段转换为大型语法树。对于每棵语法树，我们按照语句的粒度对其进行拆分，并通过先序遍历提取语句树的 token 序列。然后基于 token 序列构建语料库，并在语料库上对语句树的 token 序列做 index 映射以提取代码特征，便于后续对代码特征向量化表示。最后，将代码特征保存在服务器中。

3.6.2 流程设计

代码特征提取模块流程如图 3-17所示。该模块主要对代码的语法树做拆分处理并建立语料库和 token 序列之间的映射。具体步骤如下：

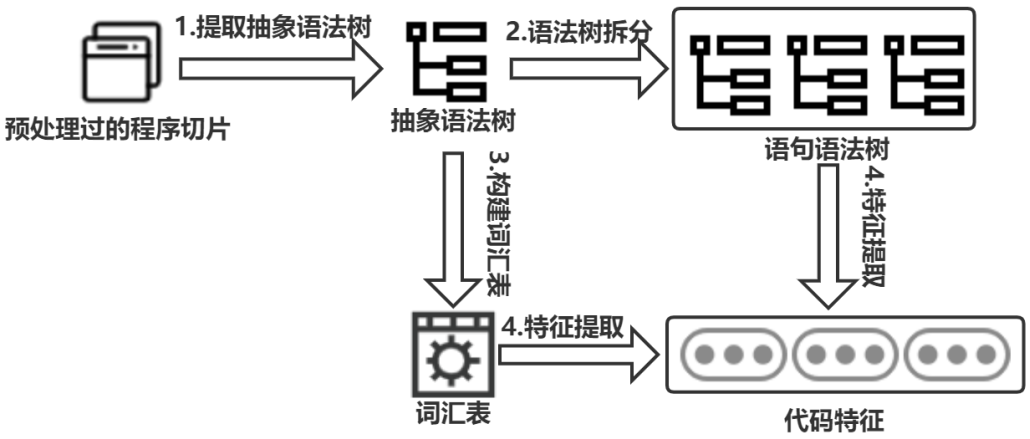


图 3-17: 代码特征提取流程图

提取抽象语法树。系统使用 javalang 将一个警告切片解析成一棵抽象语法树。javalang 不同于 JavaParser 的地方在于，javalang 解析粒度更粗，基本达到语句级别的语法树结构。而 JavaParser 更偏向于标识符粒度的语法树结构。

语法树拆分。语句语法树是指程序语言规范中定义的语句在语法树中的根节点。本系统设计了一个先序遍历算法，在语句级别上将一个代码片段的大语法树拆分为一组小的语句语法树。本系统还将 MethodDeclaration 视为一个特殊的语句节点。通过这种方式，大的语法树最终被表示为一个短的语法树集合。

构建词汇表。由于抽象语法树中有多种特殊的句法符号，本系统通过对抽象语法树的先序遍历，使用词法分析以特定的 token 表示方法将将每个代码语句分解为代码 token，得到所有的 token 作为训练语料。例如，语句“strcat(prefix, rate_str);”在经过代码预处理后被抽象为“strcat(VAR8,VAR11);”。然后将其标记为七个单独的 token：“strcat”、“(”、“VAR8”、“,”、“VAR11”、“)”和“;”。word2vec 用于学习 token 序列的无监督向量，使用 Word2Vec 在 token 序列上构建词汇表。

特征提取。本系统在每一棵语句语法树上，采取先序遍历的方式遍历语法树，并根据上一步构建好的词汇表，映射获得每个语法树节点对应的 token 索引。特征结果使用递归嵌套的方式存储，保留语法树的句法结构。

3.6.3 核心类图

如图3-18所示是代码特征提取模块的核心类图。ExtractCodeFeatureService是对外提供提取警告代码特征服务（extract_code_feature）的类。提取代码特征的具体逻辑委托 ExtractCodeFeature 来实现。

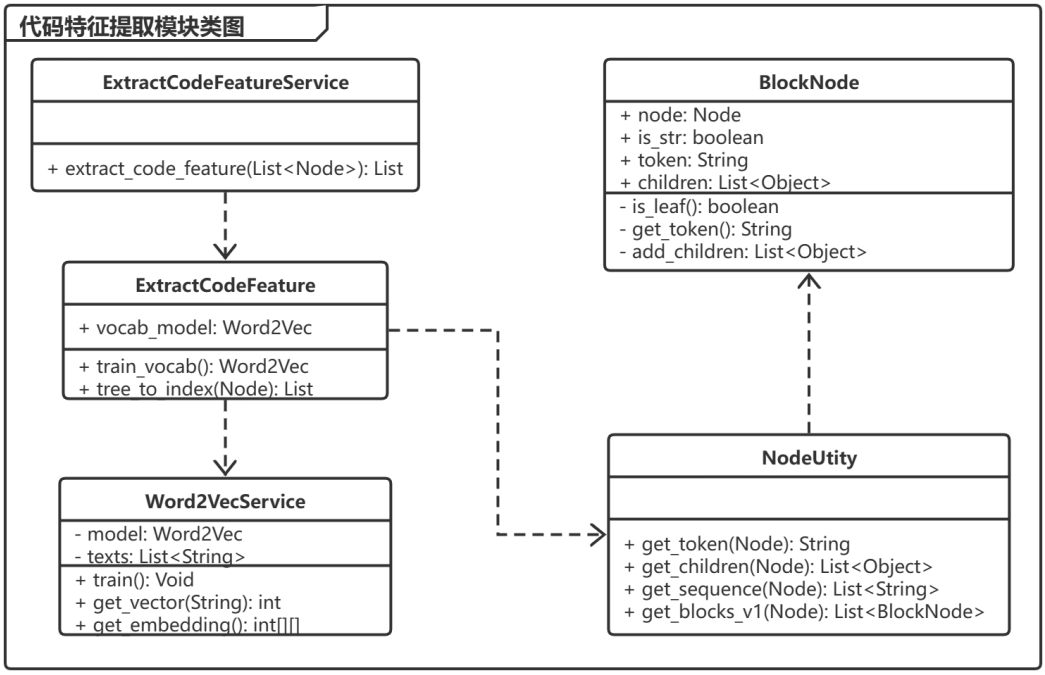


图 3-18: 代码特征提取类图

ExtractCodeFeature 是提取代码特征服务的核心类。ExtractCodeFeature 类主要有 train_vocab 和 tree_to_index 两个方法。train_vocab 方法负责构建词汇表，训练 Word2Vec 模型。Word2Vec 模型的训练以及使用逻辑封装在 Word2Vec_Service 类中，主要包含 train，get_vector 和 get_embedding 方法。train 负责基于 token 序列 texts 训练 Word2Vec 模型，训练好的模型 model 作为成员变量存储在类中。get_vector 负责获得给定 token 的向量表示。get_embedding 负责提取模型 model 中的向量矩阵，作为后一模块语句编码器的嵌入层使用。

NodeUtity 是处理抽象语法树 AST 的核心类，主要包含 get_token，get_children，get_sequence 和 get_blocks_v1 四个方法。get_token 负责获取当前 AST 节点的 token 表示。get_children 负责获取当前 AST 节点的子节点列表。get_sequence 负责提取整棵 AST 的 token 序列。get_blocks_v1 函数内会调用 get_token 和 get_children 方法，负责对 AST 做拆分处理，结果返回语句语

法树（BlockNode）集合。BlockNode 作为语句树的抽象表示，主要包含获取 token 表示（get_token），判断是否为叶子节点（is_leaf）和获取子节点列表（add_children）三个功能。BlockNode 类内部封装了该语句所对应 AST 节点的根节点，通过重写获取子节点列表（add_children）方法，去除根节点中与该语句不相关的子节点。

3.7 警告特征表示模块设计

3.7.1 架构设计

如图 3-19所示, 警告特征表示模块的主要目的是为了将警告特征的输入形式化为数字向量。警告特征由两部分组成：代码特征和手工特征。因此，首先，系统需要获得代码特征和手工特征。代码特征在代码特征提取模块已经提取出来，在该模块需应用机器学习技术将代码特征转换成向量表示；手工特征的提取过程是使用静态度量工具对警告进行计算获得一组度量值，并采用热编码，归一化等特征处理方法对度量值进行加工使其合理化。接着，采用特征融合技术将代码特征和警告特征加以拼接得到融合特征向量。最后，将融合特征持久化到服务器中。

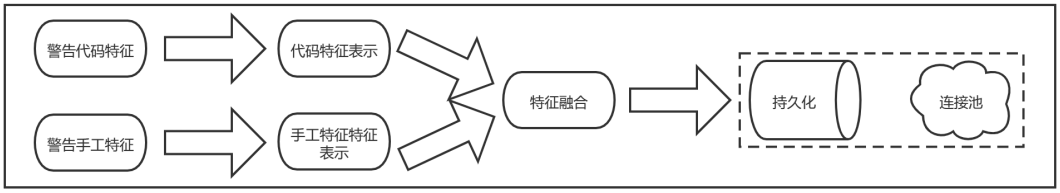


图 3-19: 警告特征表示架构图

3.7.2 流程设计

警告特征表示模块流程如图 3-20所示。由于程序切片和报告语句是词法源代码 token，而基于机器学习的警告误报分类模型要求其输入形式为数字向量。因此，需要用合适的数据结构来表示模型输入数据。本模块主要采用机器学习、特征融合等技术将警告特征转化成向量表示。具体步骤如下：

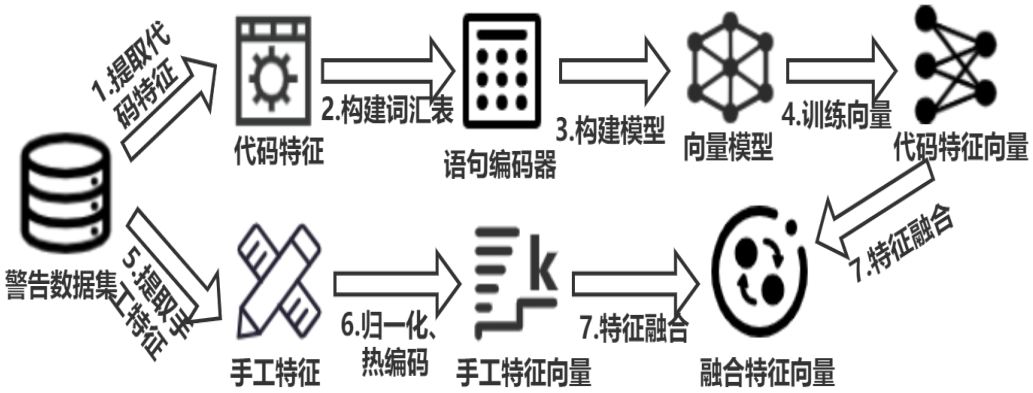


图 3-20: 警告特征表示流程图

提取代码特征。代码特征在代码特征提取模块中已经提取出来的，系统在这里需要根据自定义的路由将警告的语句语法树数组从服务器存储地址中提取出来，作为第二步语句编码器的输入。

构建语句编码器。构建语句编码器是在警告的语句语法树数组上编码语句。给定语句语法树数组，系统设计了一个语句编码器，用于学习语句的向量表示。首先需要构建词汇表。因为抽象语法树中表示标识符等词汇信息的所有叶节点也包含在语句语法树的叶节点中，所以符号嵌入可以很好地捕获词汇知识。所有语句语法树都由语句编码器编码为向量。训练后得到的符号嵌入层作为语句编码器中的初始权重参数。

构建代码特征训练模型。为了表示供警告分类器模型的输入，需要将代码特征数字向量化。构建向量训练模型是使用循环神经网络将语句和语句之间的顺序依赖编码为一个向量。这样的特征向量捕获了源代码之间的自然性，更能表示警告代码的全面特性。

训练向量。训练向量的目的是为了表示语句序列。基于语句语法树的向量序列，本系统利用 GRU 来跟踪语句的自然性。在实践中，不同警告切片中的语句数量可能会有显着差异，从而得到的语句编码表示形状不一。在实践中，语句长度可以从 3 到 102 不等。因此，为了确保所有在输入到神经网络之前都具有相同的长度，使用了填充对齐技术。为了进一步增强循环层捕获依赖信息的能力，系统采用双向 GRU，将两个方向的隐藏状态连接起来形成新状态。这些状态的最重要特征随后由最大池化或平均池化进行采样。考虑到不同语句的重要性直观上是不相等的，例如 MethodInvocation 语句中的 API 调用可能包含

更多的功能信息，因此系统默认使用 max pooling 来捕获最重要的语义。

提取手工特征。本系统提取警告报告的相关信息并使用 SourceMonitor 度量工具分析警告代码，获得圈复杂度、函数调用数等十个度量特征。经过处理后的警告报告原始的相关信息，和度量特征拼接即构成了手工特征。

归一化、热编码、数值嵌入。本系统使用归一化、热编码两种特征处理方法对手工特征做相应的处理。使用特征归一化将连续型特征的值域限制在 $[-1, 1]$；使用热编码将警告类型等离散特征转化成数字向量。警告危险等级等非连续固定值由于数值变化有限，直接采取数值嵌入即可。

特征融合。代码特征和手工特征组合在一起即为警告特征。本系统使用扁平化的方式将代码特征和手工特征的向量表示融合拼接，得到一个定长的数字化向量作为警告的特征表示。

3.7.3 核心类图

如图 3-21所示是警告特征表示模块的核心类图。ExtractAlarmFeature 是对外提供警告特征表示服务的类。ExtractAlarmFeature 类只有 merge 一个方法，负责融合代码特征表示和手工特征表示。

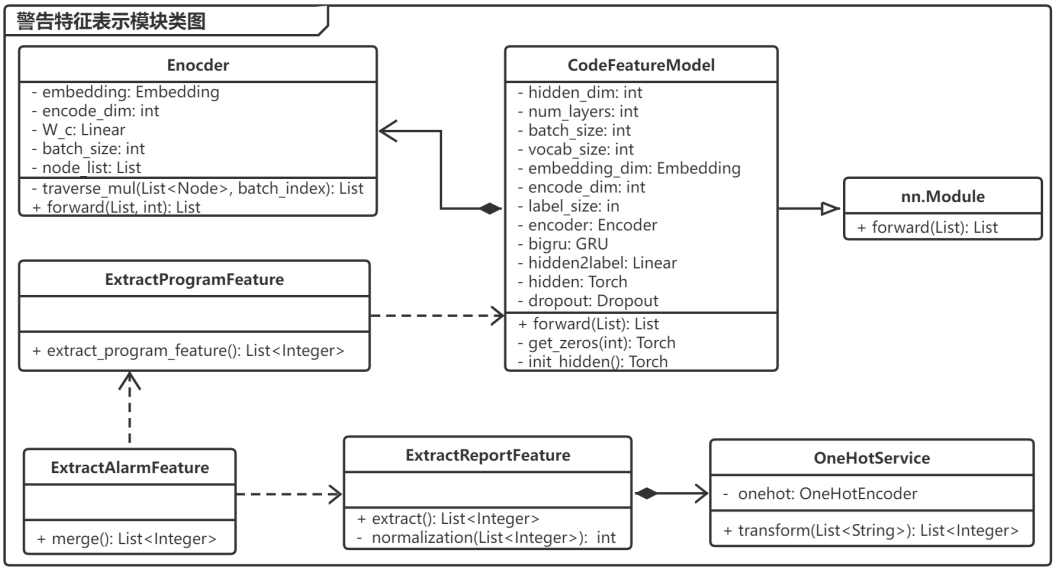


图 3-21: 警告特征表示类图

ExtractProgramFeature 是代码特征表示的核心类。ExtractProgramFeature 类主要有 extract_program_feature 一个方法。extract_program_feature 方法负责表示

代码特征，依赖 ProgramFeatureModel 类的具体实现。ProgramFeatureModel 类继承自 pytorch 框架的 nn.Module 类，是一个复杂的神经网络结构。ProgramFeatureModel 主要包含 forward，get_zeros 和 init_hidden 三个方法。get_zeros 方法负责创建全零向量，因为每个警告的语句数量不等，用来填充向量保证向量 seq_lengrh 一致。init_hidden 负责初始化隐藏层。forward 方法实现对父类函数的重写，负责训练代码特征模型，并应用模型提取向量。ProgramFeatureModel 内置了一个语句编码器（Encoder），Encoder 类负责对每棵语句语法树树进行编码，主要包含 traverse_mul 和 forward 两个方法，也继承了 nn.Module 类。forward 方法是统一对外提供语句编码的功能，traverse_mul 具体实现了批量化语句编码的逻辑。ExtractReportFeature 是手工特征表示的核心类。ExtractReportFeature 类主要有 extract_metrics_feature 和 normalization 两个方法。extract_metrics_feature 负责提取警告的度量统计特征，并调用 normalization 函数对特征做标准化处理。ExtractMetricsFeature 类内置了 OneHotService 类，负责对字符类型特征做热编码处理（transform），将字符类型转化成数值类型。

3.8 警告分类器模型构建模块设计

3.8.1 架构设计

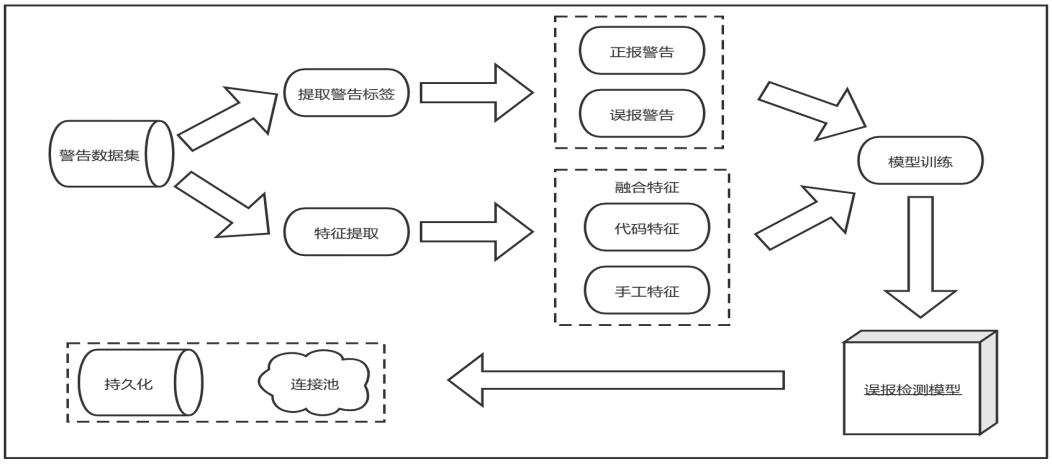


图 3-22: 警告分类器模型构建架构图

如图 3-22所示，警告分类器模型构建模块的主要目的是在警告标签和警告特征上，构建警告分类器模型。由于警告分类器模型构建耗时很长，本系统将

警告分类器模型存储到服务器上。在新的检测任务执行警告检测操作时，直接使用服务器上的模型即可，无需重复训练。

3.8.2 流程设计

警告分类器模型构建模块流程如图 3-23所示。该模块流程设计主要是基于神经网络训练模型和使用训练生成的模型来预测结果的常见方法。警告分类器模型构建的具体步骤如下：

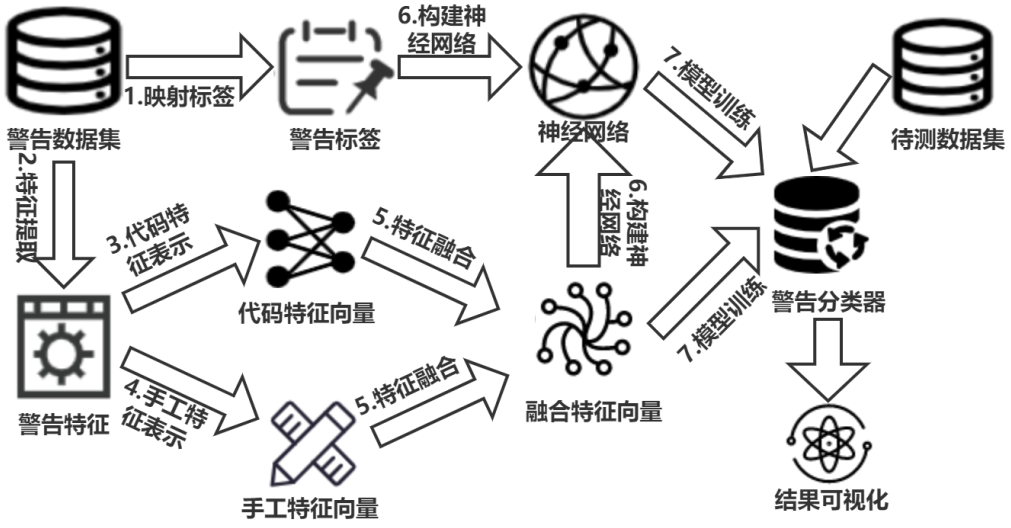


图 3-23: 警告分类器模型构建流程图

映射标签。映射标签是从警告数据集中提取警告标签并和警告特征做一一映射。警告标签有正报和误报两种分类，为了便于模型的训练，使用数字 1 表示正报标签，数字 0 表示误报标签。数字化标签值作为警告分类器模型训练的特征标签输入。

构建神经网络。构建神经网络是使用流行的深度学习框架 PyTorch[①]来搭建 GRU 循环神经网络。神经网络的主体部分是双向 GRU，GRU 的输出向量作为线性层的输入，线性层用来预测警告的标签值。为了防止预测结果的过拟合化，在线性层后连接一个非线性激活层并使用 dropout 函数屏蔽部分神经元。

模型训练。模型训练是训练警告分类器模型。在多轮训练周期中，通过损失函数计算模型预测标签和期望标签的误差值，并反向传播将梯度值存放于模

[①]https://pytorch.org/

型中。根据梯度值使用自适应学习率优化算法调整神经元之间的连接权重，以此优化神经网络参数，使得警告分类器模型能够快速收敛，并且达到分类结果准确率高的效果。

3.8.3 核心类图

如图3-24所示是警告分类器模块的核心类图。TrainDetectModel 是对外提供警告分类器构建服务的类。TrainDetectModel 类只有 train 一个方法，负责警告分类器的构建，其具体实现依赖 DetectModelBuilder。TrainDetectModel 内部聚合了代码特征表示模型（code_model）和度量特征表示模型（metrics_model）。TrainDetectModel 使用 FeatureFusion 将 code_model 提取的代码特征和 metrics_model 提取的度量特征融合在一起，DetectModelBuilder 接收警告的融合特征作为构建警告分类器模型的输入。

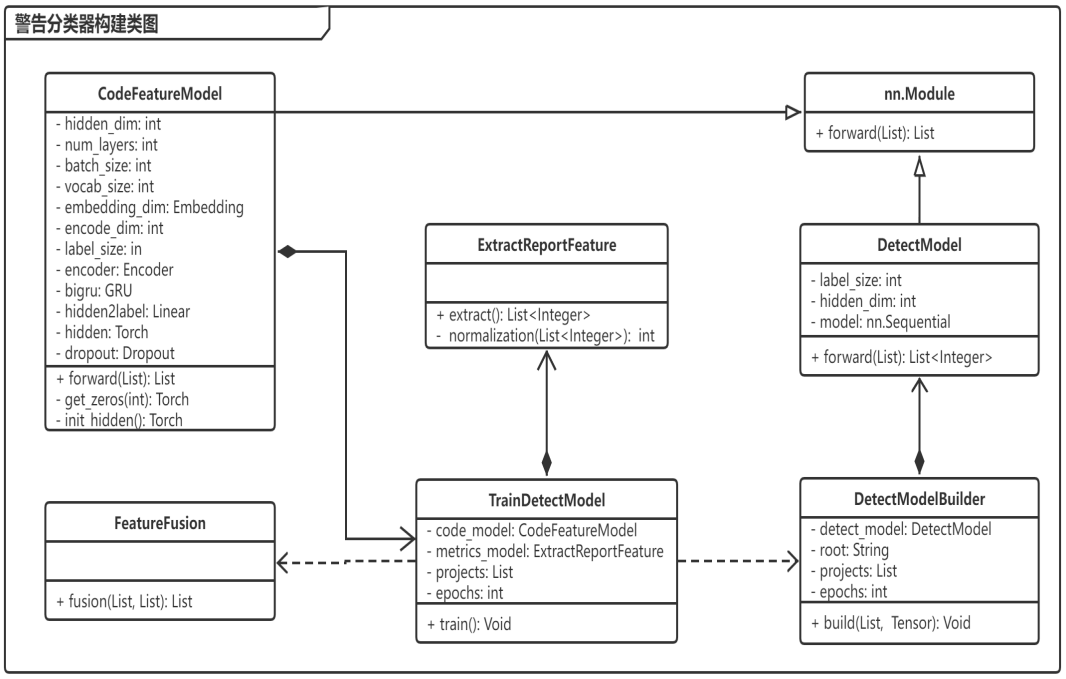


图 3-24: 警告分类器模型构建类图

DetectModelBuilder 是警告分类器模型构建的核心类。DetectModel 是系统搭建的警告分类器模型。DetectModelBuilder 类主要有 build 一个方法。build 方法负责警告分类器模型的构建，在每个训练周期内，函数内根据警告分类器 DetectModel 预测的结果，计算梯度值以优化分类器的模型参数。

DetectModel 类继承自 PyTorch 框架的 nn.Module 类，是一个复杂的神经网络结构。DetectModel 包含 forward 一个方法。forward 方法实现对父类函数的重写，负责警告分类模型的训练，并反馈警告的分类结果。

3.9 本章小结

本章首先对智能化源码警告分类系统的功能需求、非功能需求以及对应的系统用户场景进行分析；其次，对系统使用到的技术栈和编码过程使用的专业性技术进行阐述，并结合 4+1 视图从多个维度对系统进行了剖析；最后，对系统进行了模块划分，并对系统的五个核心模块：警告数据集构建模块、代码预处理模块、代码特征提取模块、警告特征表示模块和警告分类器模型构建模块，从架构设计、流程设计、核心类图三个角度切入进行了介绍、解析。

第四章　智能化源码警告分类系统的实现

本章在第三章需求分析和模块划分的基础上，对各个模块的具体实现方式进行详细的描述，主要以顺序图和关键代码作为依据来讲解。顺序图能够很好地表现模块内各部分之间的协作关系，关键代码是模块中关键部分的真实代码，可以直观地给读者展示逻辑是如何实现的。在本章的最后，会展示系统的页面，并对主要页面的功能做一些解释。

4.1　警告数据集构建模块实现

4.1.1　顺序图

如图4-1所示是警告数据集构建模块的顺序图。描述了警告数据集构建过程中各个模块是如何交互的。

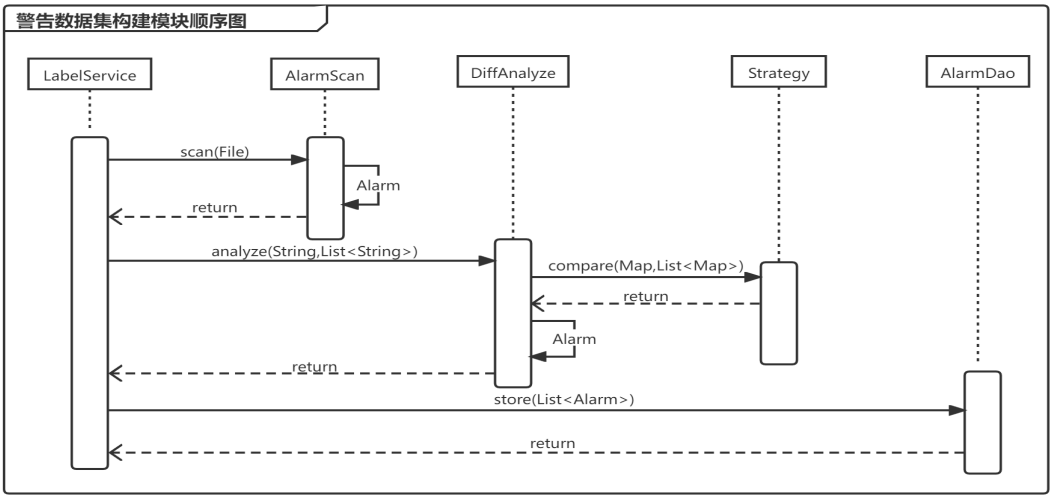


图 4-1: 警告数据集构建顺序图

在筛选并下载好 Apache 项目后，LabelService 类首先会调用 AlarmScan 类

的 scan 函数进行静态扫描。对于每一个项目的每个版本都需要执行一次扫描，扫描完成输出一份警告报告，提取警告信息得到警告初始列表并持久化进数据库中。获得初始警告列表后，便可开始警告标记工作。LabelService 调用 DiffAnalyze 类的 analyze（label）函数根据差异分析法对初始警告打上标签，具体是将待标记版本与排序之后的版本的初始警告做比对，如果在与某个版本的比对过程中未找到同样的警告，则标为正报；否则标为误报。DiffAnalyze 类会调用 Stategy 类的 compare 函数比对待标记版本的初始警告，比对策略有三种：基于位置的比对策略、基于文本的比对策略、基于哈希的比对策略。Strategy 类分别由 LocationStrategy 类、ContextStrategy 类、HashStrategy 类具体实现，DiffAnalyze 类分别应用三种比对策略标记警告，具体来说，警告应用三种比对策略全部标记为正报，标签才为正报；否则即为误报，标记完成获得警告数据集。LabelService 类调用 AlarmDao 类的 store 函数将警告数据集持久化到 MySQL 中。

4.1.2 关键代码

警告数据集构建模块最关键的地方是如何标记警告的标签，本系统采用的标记策略是差异分析标记法。差异分析标记法中最重要的部分是判定两个警告是否为同一个警告，即警告比对策略。警告比对策略中最为复杂的是基于位置的比对策略，图4-2展示了应用基于位置的警告比对策略标记警告列表的关键代码。应用基于位置的警告比对策略标记警告列表的函数入口是 label 函数，label 接受的两个参数是 Map 和 List<Map>，前者是待标记版本的警告列表，后者是对比版本的警告列表，map 的 key 值是项目中的某个文件名，value 值是该文件中的警告列表，即通过一个 map 数据结构表示了项目的某个版本中所有的警告。应用基于位置的警告比对策略标记警告列表的具体实现是这样的：基于位置的警告比对策略是对同名文件中所属同区块的警告进行比对，因此 List<AlarmDo> 存储的是经过区块划分的警告列表，警告间保持着出现在源文件中位置的先后顺序。对于待标记版本中的每个警告，调用 isSame 函数（使用基于位置的比对策略）和对比版本同名文件中同位置的警告进行对比，使用 count 变量计数警告出现在对比版本集中的次数。初始警告默认是正报，当 count 变量的值等于对比版本集的大小，即证明该警告在所有对比版本中都出现了。根据差异分析标记法原理，将其标记为误报。

isSame 函数是基于位置的警告比对策略的具体实现，其作用是比对两个警

告是否一致，接受的两个参数是待标记警告和对比警告，返回值是布尔值。函数内具体逻辑是这样的：首先，比对两个警告的类型是否一致，不一致则直接判警告不同；接着，读取两个警告的源代码代码片段文本，并计算警告的代码片段行数；最后，根据两个警告是否为匹配对（代码片段内容是否相同），采取两种不同的比对方式。如果为匹配对，证明源文件该位置的警告代码没有发生改动，代码片段行数一致则代表警告相同；如果为非匹配对，说明源文件该位置的警告代码发生了改动（增删改几行），那么代码片段行数差在给定阈值的范围内，则表明警告相同。本系统还应用了其他两种警告比对策略对警告做差异分析标记，尽最大可能过滤掉相同警告，提升数据集标签的质量。

```
@Override
public void label(Map<String, List<AlarmDO>> targetMap, List<Map<String, List<AlarmDO>>>
standardMaps) {
    targetMap.forEach((targetFile , targetDOs) -> {
        for (int i = 0 ; i < targetDOs.size() ; i ++) {
            int count = 0;
            for (Map<String, List<AlarmDO>> standardMap : standardMaps) {
                List<AlarmDO> standardDOs = standardMap.get(targetFile);
                if (standardDOs == null || i > standardDOs.size() - 1) {
                    break;
                }
                if (isSame(targetDOs.get(i) , standardDOs.get(i))) {
                    count ++;
                }
            }
            if (count == standardMaps.size()) {
                targetDOs.get(i).setPositive(false);
            } } });
}
private boolean isSame(AlarmDO targetDO , AlarmDO standardDO) {
    if (!targetDO.getType().equals(standardDO.getType())) {
        return false;
    }
    List<String> targetContents = FileTool.readRangeLine(targetDO.getAbsolutePath() ,
targetDO.getStart() , targetDO.getEnd());
    List<String> standardContents = FileTool.readRangeLine(standardDO.getAbsolutePath() ,
standardDO.getStart() , standardDO.getEnd());
    int distance1 = targetDO.getEnd() - targetDO.getStart();
    int distance2 = standardDO.getEnd() - standardDO.getStart();
    if (targetContents.retainAll(standardContents)) {      // 匹配对
        return distance1 == distance2;
    }
    else { // 非匹配对
        return Math.abs(distance1 - distance2) <= Threshold;
    }
}
```

图 4-2: 警告数据集构建关键代码

4.2 代码预处理模块实现

4.2.1 顺序图

如图4-3所示是代码预处理模块的顺序图。该模块顺序图展示了代码预处理过程中提取程序切片、提取语法树做代码补全和标识符抽象的交互流程。

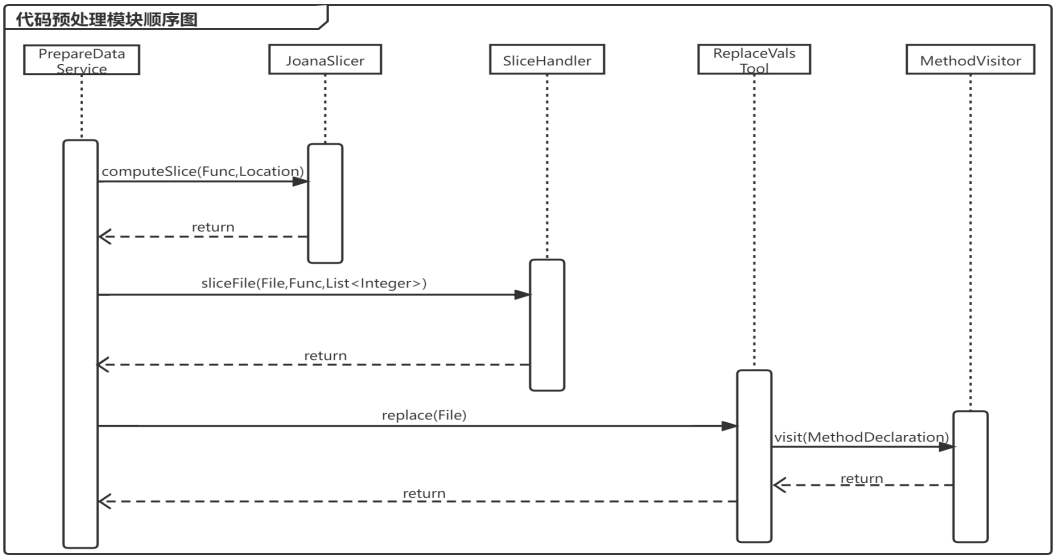


图 4-3: 代码预处理顺序图

PrePareDataService 类是向外提供服务的统一接口。PrePareDataService 类首先调用 JoanaSlicer 类的 computeSlice 函数对警告提取程序切片，JoanaSlicer 类内部封装了 Joana 静态分析工具，借助 Joana 依次构建系统依赖图（SDG）、设置入口函数、定位切片点、计算程序切片，获得程序切片初始行号结果；然后调用 SliceHandler 类的 sliceFile 函数对初始程序切片结果进行代码补全，从而达到程序切片可编译的效果。SliceHandler 类借助 JavaParser 类提取警告行所在函数的抽象语法树（AST），根据抽象语法树的语法结构，保留初始行号范围内的语法树节点，再将修改后的抽象语法树逆向转换成源代码的形式；最后调用 ReplaceValsTool 类的 replace 函数对程序切片中的标识符做抽象化处理。ReplaceValsTool 类调用 MethodVisitor 的 visit 函数，采用 JavaParser 特定的深度遍历方式，对特定的语法树节点进行数值抽象化，再将抽象化的语法树转换成源代码。通过以上步骤，不仅能将警告代码缩减成保持其语义的最小形式，还弱化因编程习惯、程序复杂度等带来的命名空间不一致而导致的无关代码信

息，极大地降低了警告代码的噪声。

4.2.2 具体实现

程序切片预处理过程中的代码补全算法如下4.1所示。我们基于抽象语法树拥有保证代码语法正确可编译的特性下，通过删除与警告无关语法树节点，以达到对警告切片剪枝的目的。

算法 4.1 使用 AST 补全代码算法

```
1:  for node in nodeList do
2:    if node.getRange().isPresent() then
3:      start ← node.getRange().get().begin.line;
4:      end ← node.getRange().get().end.line;
5:      flag ← false;
6:      for line in lines do
7:        flag ← true;
8:        break;
9:      end for
10:     parent ← node.getParentNode();
11:     if parent.isPresent() and parent instanceof SwitchEntry and node == par-
        ent.getChildNodes().get(0) then
12:       flag ← true;
13:     end if
14:   end if
15:   temp ← node;
16:   while temp.getParentNode().isPresent() do
17:     if temp instanceof Parameter and temp.getParentNode().isPresent() and
        temp.getParentNode() instanceof CatchClause then
18:       flag ← true;
19:       break;
20:     end if
21:   end while
22:   if !flag then
23:     node.remove();
24:   end if
25: end for
```

该算法的输入是警告所在函数的语法树节点和初始程序切片行号列表。首先，获取函数的方法体 body 节点并调用 walk 函数对方法体节点做层次化遍历，得到方法体的子节点列表，层次化顺序的子节点列表，使得语法树是从外到内开始删除的，不仅能提高语法树修改的效率（父节点删除则子节

点也随之删除，无需再考虑子节点），还能降低语法树修改出错率。接着，根据初始程序切片行号列表，对上一步得到的方法体子节点列表筛选删除。一般而言（除了 SwitchEntry 和 Parameter 类型节点），每个 AST 节点都可以映射到源代码中，获得其在源代码的始末行号，即为该节点的作用范围。遍历初始程序切片行号列表，只要其中一行落入该节点的作用范围内，则需要保留，保证语法完整性。至于 SwitchEntry 和 CatchCause 类型节点，需要特殊处理。SwitchEntry 类型节点对应于源代码中的 case 语句，例如 case 1:return，是由 case 节点（case），条件值节点（1）和 BlockStmt 节点（return）组成的。考虑到这种情况：case 1:return，return 行在初始行号列表中，case 节点的作用范围包括了 return 行会保留下来，而条件值节点作为 SwitchEntry 节点的子节点在进行筛选时，因为其作用范围只在 return 行的上一行，如果不特殊处理会被删除，处理后的结果会变成：default:return（条件值为空从语法树转换成源代码为 default），尽管没有破坏其语法特征，但是源代码会失去原有的语义。因此，本算法对未删除的 SwitchEntry 节点的首个子节点（条件值节点）做保留。CatchCause 类型节点对应源代码中的 catch 语句，例如 catch(IOException e)System.out.println();，是由 Parameter 节点（IOException e）和 BlockStmt（System.out.println()）节点构成的。CatchCause 节点的处理和 SwitchEntry 节点有着异曲同工之妙。假设 System.out.println() 在初始列表中，同样地，Parameter 作为 CatchCause 节点的子节点，如果不经过特殊处理，则会被误删处理。这种情况带来的影响比前者更大，因为会破坏整个语法树的语法结构，从而导致整棵语法树不可编译，无法转换成源码形式。因此，本算法对未删除的 CatchCause 节点的 Parameter 节点做保留；最后，本算法对标记需要删除的节点进行 remove，并将修改的语法树转换成源代码的形式。

```
private static void doCopyFile(final File srcFile, final File destFile, final boolean preserveFileDate)
throws IOException {
        if (destFile.exists() && destFile.isDirectory()) {
        final long srcLen = srcFile.length();
        final long dstLen = destFile.length();
        if (srcLen != dstLen) {
        if (preserveFileDate) {
            destFile.setLastModified(srcFile.lastModified());
    }
```

(a) 初始程序切片结果

```
private static void doCopyFile(final File srcFile, final File destFile, final boolean preserveFileDate)
throws IOException {
        if (destFile.exists() && destFile.isDirectory()) {
        }
        final long srcLen = srcFile.length();
        final long dstLen = destFile.length();
        if (srcLen != dstLen) {
        }
        if (preserveFileDate) {
            destFile.setLastModified(srcFile.lastModified());
        }
    }
```

(b) 代码补全切片结果

图 4-4: 程序切片补全

代码补全后的效果如图4-4所示。图4-4(a)是初始程序切片结果，很明显该切片不满足可编译特性，基于此程序切片是无法提取抽象语法树的。在经过本系统的代码补全算法处理后，得到了如图4-4(b)所示的程序切片结果。

4.2.3 关键代码

VoidVisitorAdapter 是 JavaParser 提供的一种语法树访问方式，类中定义了每种类型的节点如何遍历其子节点的顺序，通过该遍历器可以完整地遍历所有类型的语法树节点。该类作为抽象类可以很方便地给使用者做定制化开发。使用者采用继承的方式，只需专注于特定节点的处理逻辑实现；在处理逻辑后调用 VoidVisitorAdapter 类的 visit 函数，VoidVisitorAdapter 类的 visit 函数会回调其子类的 visit 函数，从而实现节点遍历顺序和节点处理逻辑的代码分离。

本系统中的 MethodVisitor 类是处理标识符抽象的核心类。MethodVisitor 类继承自 VoidVisitorAdapter 类，主要处理 NameExpr、StringLiteralExpr、VariableDeclarator、Parameter、IntegerLiteralExpr 五种类型的节点，五种类型节点的详细描述如表 4-1所示。

表 4-1: JavaParser 节点类型信息表

Type	Desc	Example
IntegerLiteralExpr	数值型字面值	**8934**
NameExpr	使用变量名	int x=**a** + 3
Parameter	入口函数参数	int abc(**String x**)
StringLiteralExpr	字符串字面值	**“Hello World”**
VariableDeclarator	申明变量	**int x=14**

MethodVisitor 类的关键代码如图 4-5所示。MethodVisitor 类分别处理 NameExpr、StringLiteralExpr、VariableDeclarator、Parameter、IntegerLiteralExpr 五种类型的节点。由于图片太大页面放不下的原因，故图 4-5中略去了 IntegerLiteralExpr 类型节点的处理逻辑。每种类型节点的处理函数名都为 visit，即每次处理调用只会处理一种类型的节点。父类 VoidVisitorAdapter 封装了所有类型节点的处理逻辑（遍历方式），通过 super.visit 回调来实现对 AST 节点的全面遍历。处理 NameExpr、VariableDeclarator 和 Parameter 类型节点，MethodVisitor 类内部维护了一个变量名列表 vals，如果 NameExpr 节点的名字在 vals 中未出现，更新 vals 列表；否则，直接使用“val”+ index 的命名方式替换节点命名。处理 StringLiteralExpr 类型节点，MethodVisitor 类内部维护了一个字符串字面值列表 strs，如果 StringLiteralExpr 节点的值在 strs 中未出现，更新 strs 列表；否则，直接使用“str”+ index 的命名方式替换节点值。处理 IntegerLiteralExpr

```
public void visit(NameExpr expr, List<String> result) { //使用变量名
    String valName = expr.getName().asString();
    if (vals.contains(valName)) {
        result.add("val" + (vals.indexOf(valName) + 1));
        expr.setName("val" + (vals.indexOf(valName) + 1));
    }else {
        vals.add(valName);
        result.add("val" + vals.size());
        expr.setName("val" + (vals.indexOf(valName) + 1));
    }
    super.visit(expr, result);
}
public void visit(StringLiteralExpr expr, List<String> result) { //字符串字面值
    String strValue = expr.getValue();
    if (strs.contains(strValue)) {
        result.add("str" + (strs.indexOf(strValue) + 1));
        expr.setEscapedValue("str" + (strs.indexOf(strValue) + 1));
    }else {
        strs.add(strValue);
        expr.setEscapedValue("str" + strs.size());
    }
    super.visit(expr, result);
}
public void visit(VariableDeclarator declarator, List<String> result) { //变量申明
    String valName = declarator.getName().asString();
    if (vals.contains(valName)) {
        declarator.setName("val" + (vals.indexOf(valName) + 1));
    }else {
        vals.add(valName);
        declarator.setName("val" + vals.size());
    }
    super.visit(declarator, result);
}
public void visit(Parameter parameter , List<String> result) { //函数参数
    String valName = parameter.getName().asString();
    if (vals.contains(valName)) {
        parameter.setName("val" + (vals.indexOf(valName) + 1));
    }else {
        vals.add(valName);
        parameter.setName("val" + vals.size());
    }
    super.visit(parameter, result);
}
```

图 4-5: 代码预处理关键代码

类型节点，得到节点的数值判断正负并计算数值的位数。不同的位数采取如下方式表示：（1）. 一位数保留原数值；（2）. 两位正数表示为 P2，负数表示为 N2；（3）. 三位正数表示为 P3，负数表示为 N3。（4）. 四位及以上正数表示为 P4+，负数表示为 P4-。通过标识符抽象处理，可以去除因开发者编程习惯或程序复杂度导致的代码风格差异，一定程度上减少了语料库的词汇数量，大大提升了代码特征的质量。

4.3 代码特征提取模块实现

4.3.1 顺序图

如图4-6所示是代码特征提取模块的顺序图。该模块顺序图展示了抽象语法树拆分、构建语料库的调用流程，是本系统的核心部分。

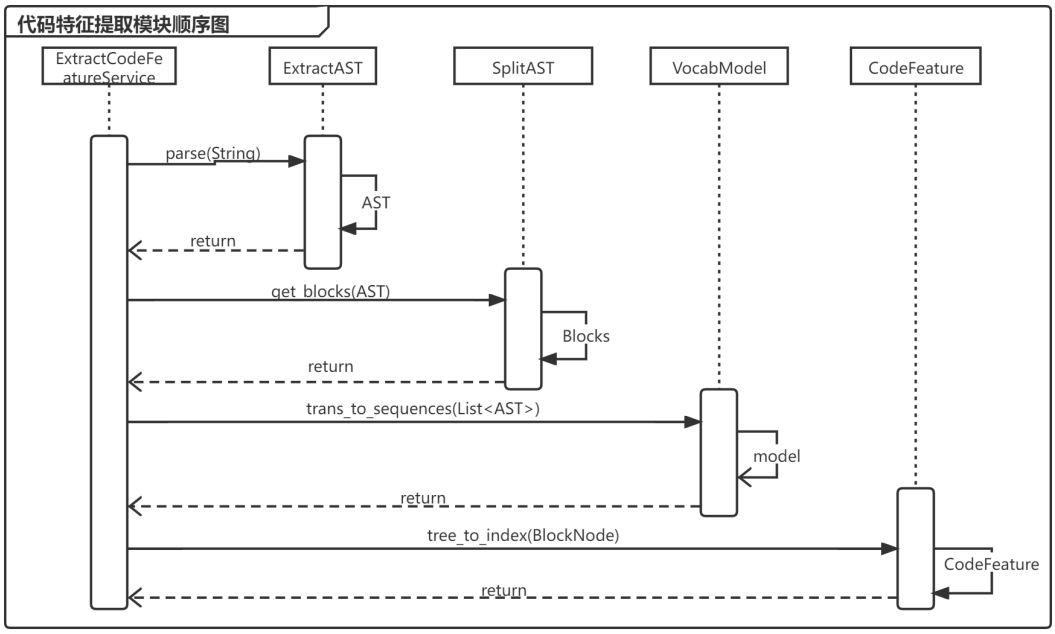


图 4-6: 代码特征提取顺序图

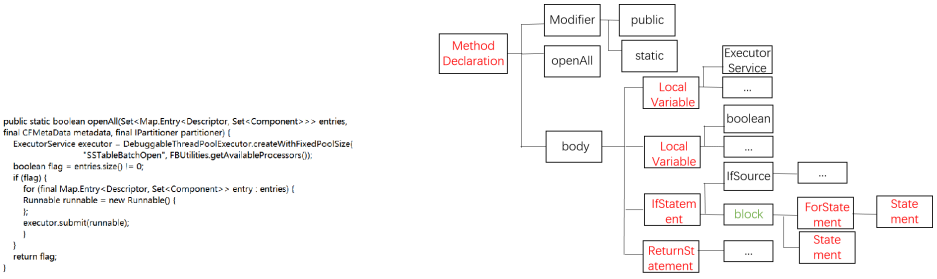
ExtractCodeFeatureService 类是该模块的入口，是对外统一提供服务的接口。ExtractCodeFeatureService 类首先调用 ExtractAST 类的 pase 函数提取代码预处理后得到的程序切片的抽象语法树，借助 javalang 提取；然后调用 SplitAST 类的 get_blocks 函数对程序切片的抽象语法树进行拆分，具体来说，

系统是按照语句级别的粒度对语法树进行分解；接着调用 VocabModel 类的 trans_to_sequences 函数构建语料库；最后调用 CodeFeature 类的 tree_to_index 方法提取代码特征。

4.3.2 具体实现

形式上，给定一棵语法树 T 和一组 Statement 语法树节点 S，T 中的每个语句节点 $s \in S$ 对应一个源代码语句。本文将 MethodDeclaration 视为一个特殊的 Statement 节点，因此 $S = S \cup \{MethodDeclaration\}$。对于嵌套语句，如图 4-7所示。系统定义了一组单独的节点 $P = \{block, body\}$，其中 $block$ 是嵌套语句（例如 Try 和 While 语句）的头部和主体，需要进行拆分，而 $body$ 是函数申明。语句节点 $s \in S$ 的所有后代都用 $D(s)$ 表示。对于任何 $d \in D(s)$，如果存在一条经过节点 $p \in P$ 的从 s 到 d 的路径，则意味着节点 d 包含在语句 s 主体中的一条语句中，即节点 d 为 s 的子语句节点。一个以语句节点 s 为根的语句树是由节点 s 和它的所有后代组成的，但是不包括它在 T 中的子语句节点。例如，以 MethodDeclaration 为根的第一棵语句树在图 4-7(b)中用红色标出，其中包括“static”、“public”和“openAll”等头部部分，并排除了主体中两个 LocalVariable、一个 If 和一个 Return 语句节点。由于一棵语句树的节点可能由三个或者更多的子节点，所以语句树相较于二叉树而言，其实是一棵多路树。通过这种方式，一棵大型的语法树可以分解为一系列不重叠的多路语句树。

语法树的拆分和语句树序列的构造通过遍历器和构造函数实现。遍历器以先序遍历的方式访问语法树的每个节点，构造函数递归地创建一个语句树并顺序添加进语句树序列中。这种做法保证了按照源码中的语句顺序附加一棵新的语句树。最终可以得到语句树序列作为代码特征训练模型的原始输入。



(a) 源代码 (b) AST 语法结构

图 4-7: 语法树拆分

语法树拆分算法的具体实现如算法 4.2所示。整个语法树拆分算法总体来说

是一个先根遍历、递归分解的一个过程。整个算法涉及到的函数调用比较多，下面先一一描述算法中出现的一些函数以及数据结构。get_token 函数接受的参数是语法树节点，负责提取当前节点的 token。前面的章节也讲到，javalang 提取出来的语法树是粗粒度的，基本是按照语句粒度来划分节点的，因此节点会包含其他数据结构类型。经研究发现，语法树的结构是由节点 node、集合 set 和字符串 str 类型构成的。节点 node 即对应语句在语法树中的根节点，集合 set 一般是函数申明里面的参数、修饰符组成的列表，字符串 str 往往会是一些变量名。针对不同类型的节点，get_token 函数是按照如下约束提取 token 的：（1）.node 类型节点的 token 值为具体类名（TryStatement 等）；（3）.set 类型节点的 token 值为“Modifier”；（2）. 字符串类型节点的 token 值为自身。get_token 函数在遇到 set 类型子节点时，需要做 expand 特殊处理，将 set 中的每个节点合并到子节点列表中，不作为一个整体存在，方便后续递归调用算法。BlockNode 是本算法的一个重要数据结构，其表示的是一棵拆分出来的语句树。BlockNode 封装了拆分出来的语句树根节点，最关键的地方是重写了语法树 node 节点的获取子节点列表的方式。根据不同类型节点的内部结构，重写 Node 节点的 get_child 方法，以保证语句树不包含和当前语句不相关的其他节点。

Java 程序主要由逻辑语句、代码块、类与函数申明和顺序语句这四种类型语句组成。而前三种类型语句都是由多行代码组成的，因此本算法围绕这三种类型语句作为切入口，采取先根访问的遍历顺序，由外而内层层拆分整棵抽象语法树。算法初始会提取该节点的 token 值和子节点列表，算法根据 token 值判断节点类型，并对子节点列表进行递归调用。算法的处理逻辑是这样的：（1）. 当前节点属于类与函数申明类型节点，算法首先会将当前节点封装成一棵语句树；接着获得该节点的主体部分，并对主体中的所有子节点进行分情况处理：如果子节点不是逻辑语句节点并且不是代码块（顺序语句集合），则表明该节点是一条语句，将其封装成一颗语句树；其他情况则表明子节点结构较为复杂，递归调用算法拆分该节点。（2）. 当前节点属于逻辑语句，算法首先会将当前节点（if、for 等）封装成一棵语句树；由于逻辑语句类型节点的第一个子节点代表的是逻辑语句中的表达式信息，因此只需从第二个子节点做拆分处理。同样分为两种情况：如果子节点不是逻辑语句或者代码块类型的节点，则只是普通的语句，将其封装成一颗语句树；其他情况则递归调用算法做拆分。（3）. 当前节点是代码块类型节点，处理逻辑基本和第二种情况一致。

算法 **4.2** *getblocks*(*node*, *blockseq*)

```
1: name, children ← gettoken(node), getchildren(node);
2: logic ← ['SwitchStatement','IfStatement', 'ForStatement', 'WhileStatement',
   'DoStatement'];
3: if name in ['MethodDeclaration', 'ConstructorDeclaration'] then
4:     blockseq.append(BlockNode(node));
5:     body = node.body;
6:     for child in body do
7:         if gettoken(child) not in logic and not hasattr(child, 'block') then
8:             blockseq.append(BlockNode(child));
9:         else
10:            getblocks(child, blockseq);
11:        end if
12:    end for
13: else if name in logic then
14:    blockseq.append(BlockNode(node));
15:    for child in children[1:] do
16:        token ← gettoken(child);
17:        if not hasattr(node, 'block') and token not in logic+['BlockStatement'] then
18:            blockseq.append(BlockNode(child));
19:        else
20:            getblocks(child, blockseq);
21:        end if
22:        blockseq.append(BlockNode('END'));
23:    end for
24: else if name is 'BlockStatement' or hasattr(node, 'block') then
25:    blockseq.append(BlockNode(name));
26:    for child in children do
27:        if gettoken(child)not in logic then
28:            blockseq.append(BlockNode(child));
29:        else
30:            getblocks(child, blockseq);
31:        end if
32:    end for
33: else
34:    for child in children do
35:        getblocks(child, blockseq);
36:    end for
37: end if
```

4.3.3 关键代码

代码特征提取的关键代码如图4-8所示。代码特征提取主要由 vectorlize 和 train_vocab 两个函数相互协作实现完成的。train_vocab 函数是用于训练语料库模型的，内部嵌有 trans_to_sequences 函数，trans_to_sequences 接受的参数是一棵抽象语法树，调用 get_sequence 函数采用先序遍历的方式，使用特定的节点序列表示法，将抽象语法树转化成 token 序列。使用 trans_to_sequences 对训练数据集中的每棵抽象语法树提取 token 序列，得到语料库 corpus。在语料库 corpus 基础上，使用 Word2Vec 模型训练语料库模型，并将语料库模型持久化在服务器中。

```
def vectorlize(self, batch_data):
    vocab = self.vocab_model.wv.vocab
    max_token = self.vocab_model.wv.syn0.shape[0]
    def tree_to_index(node):
        token = node.token
        result = [vocab[token].index if token in vocab else max_token]
        children = node.children
        for child in children:
            result.append(tree_to_index(child))
        return result
    def trans2seq(r):
        blocks = []
        func(r, blocks)
        tree = []
        for b in blocks:
            btree = tree_to_index(b)
            tree.append(btree)
        return tree

def train_vocab(self):
    def trans_to_sequences(ast):
        sequence = []
        get_sequence(ast, sequence)  #获得 ast 的 token 序列,ast 即 FileAST 根节点
        return sequence
    corpus = []
    for node in self.train_data:
        corpus.append(trans_to_sequences(node))
    w2v = Word2Vec(corpus, size=128, workers=16, sg=1, min_count=3)
    self.vocab_model = w2v
```

图 4-8: 代码特征提取关键代码

vectorlize 函数批量化提取警告代码特征，接受的参数是批量 AST（AST 数组列表）。函数中包含 tree_to_index 和 trans2seq 两个内部方法，trans2seq 是

vectorlize 提取警告代码特征的主要方法。对于每一棵 AST，调用 func 函数对其做语句级别拆分，得到语句 AST 列表。从理论上讲，blocks 的大小即为语句的个数。对于每棵语句 AST，执行 tree_to_index 函数，应用 trans_to_sequences 训练得到的语料库模型，建立 token 和语料库中索引 index 的映射关系。通过列表中嵌套列表的方式，达到每棵语句 AST 中的父子节点都保持层级结构的效果。最终得到一组根据语句 token 映射到语料库 index 的层次化结构数组，便于后续对代码特征做数字化向量处理。

4.4 警告特征表示模块实现

4.4.1 顺序图

如图4-9所示是警告特征表示模块的顺序图。该模块顺序图展示了代码特征表示、度量特征表示和特征融合的交互过程，是本系统核心模块之一。

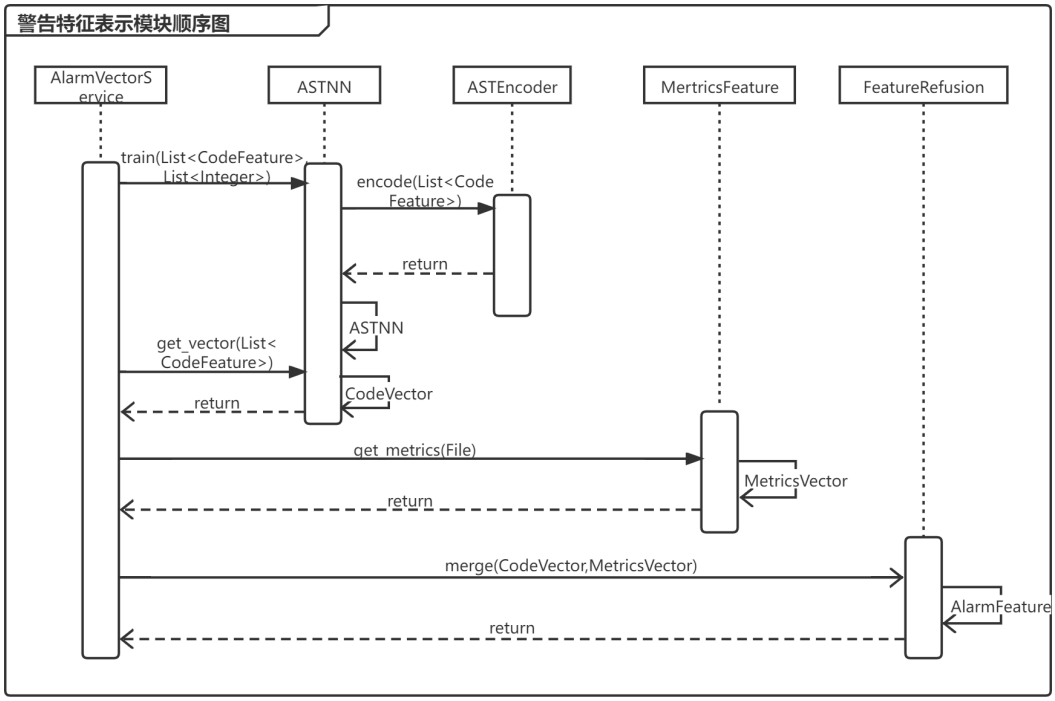


图 4-9: 警告特征表示顺序图

该模块的入口是 AlarmVectorService 类，负责统一对外提供服务。AlarmVectorService 类首先调用 ASTNN 类的 train 函数，用来训练警告的代码特

征，得到代码特征提取模型，代码特征模型会持久化到服务器上。ASTNN 类会调用 ASTEncoder 类的 encode 函数，ASTEncode 作为语句编码器，负责将 AST 中每条子语句 AST 编码为向量，每个 AST 就被编码成了二维矩阵，ASTNN 类借助神经网络的能力可以将二维矩阵转换成一维向量；然后调用 ASTNN 类的 get_vector 函数获得代码特征向量。ASTNN 类应用训练好的代码特征提取模型，将代码特征数字化表示；接着调用 MetricsFeature 的 get_metrics 函数，使用 SoureMonitor 分析警告，获得警告的度量特征；最后，调用 FeatureRefusion 类的 merge 函数，使用扁平化的方式将代码特征和度量特征加以融合得到完整的警告特征表示。

4.4.2 具体实现

本系统使用神经网络来提取代码特征向量，其训练过程如图 4-10所示。

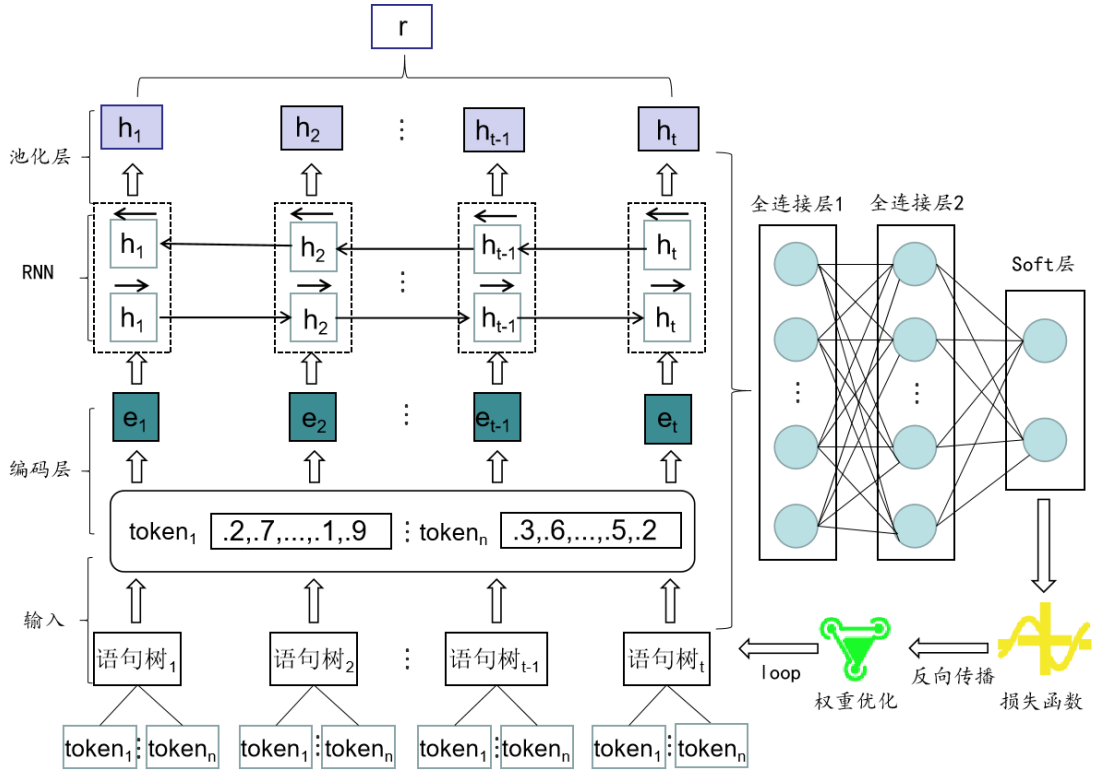


图 4-10: 代码特征模型训练过程

（1）. 给定语句语法树，设计了一个基于 embedding 层的语句编码器，用于学习语句的向量表示。embedding 层的初始权重来源于代码特征提取模块训练的语料库模型。

（2）. 使用语句编码器得到每条语句的向量表示，每棵抽象语法树由多个定长语句向量组成二维矩阵。在批量化处理过程中采用对齐填充的方式做标准化处理，利用双向 GRU 神经网络来跟踪语句的自然性，学习语句间的顺序关系并得到新的向量序列。

（3）. 使用双向 GRU 获得语句序列的最重要特征后，采用 max pooling 来捕获最重要的语义并对向量序列进行转置和降维得到抽象语法树的一维向量表示。池化后产生的向量，被视为代码片段的初始向量表示。

（4）. 使用两个全连接层对代码向量做长度的变化后，应用 SoftMax 计算向量在二分类上的概率。在多轮训练周期中，使用损失函数比对预测标签和实际标签的差异并反向传播，优化器根据传播结果优化网络参数权重直至模型完全收敛。

总体而言，该代码特征模型的结构主要由编码层、循环层、池化层、全连接层和 SoftMax 层组成，经过多轮训练周期学习后，即可应用该模型提取警告的代码特征向量。

4.4.3 关键代码

警告特征表示模块最核心的部分是代码特征的表示。代码特征表示的关键代码如图4-11所示。代码特征表示是通过代码特征表示模型训练进而应用获得代码特征向量的。在本系统中，由 ProgramFeatureModel 类负责具体的实现。ProgramFeatureModel 类内置了一个语句编码器，由 Encode 类负责语句编码的具体实现。

Encoder 类需要词汇数量（vocab_size）、嵌入层维度（embedding_dim）、编码维度（encode_dim）和初始权重（pretrained_weight）这四个参数进行初始化。Encoder 本质上也是神经网络结构，内部封装了嵌入层（embedding）。forward 函数是 Encoder 类语句编码的入口函数，具体编码逻辑委托 traverse_mul 函数实现。traverse_mul 批量化编码语句树，具体来说会同时处理 batch_size 数量的 ST 树。traverse_mul 函数是一个递归调用的过程。

每一次调用，对每棵语句树的相同位置节点进行编码并和该语句树已编码得到的向量进行累加。在 traverse_mul 函数内部维护一个 child 二维数组（由于篇幅所限，代码中未呈现），负责记录每个位置的子节点列表。借助 child 数组，可以精确有效地递归执行 traverse_mul 函数，提升语句编码的总体效率。

```
class Encoder(nn.Module):
    def __init__(self, vocab_size, embedding_dim, encode_dim,pretrained_weight):
        super(Encoder, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.encode_dim = encode_dim
        self.node_list = []
        self.embedding.weight.data.copy_(torch.from_numpy(pretrained_weight))
    def traverse_mul(self, node, batch_index):
        for c in range(len(children)):
            tree = self.traverse_mul(children[c], batch_children_index)
            if tree is not None:
                batch+=zeros.index_copy(torch.LongTensor(children_index[c])), tree)
        return batch
    def forward(self, x, batch_size):
        self.batch_node=self.create_tensor(torch.zeros(batch_size, self.encode_dim)))
        self.node_list=self.traverse_mul(x, list(range(batch_size)))
        return self.node_list
class ProgramFeatureModel(nn.Module):
    def __init__(self, embedding_dim, hidden_dim, vocab_size, encode_dim):
        super(ProgramFeatureModel, self).__init__()
        self.encoder= Encoder(vocab_size,embedding_dim,encode_dim,pretrained_weight)
        self.bigru=nn.GRU(encode_dim,hidden_dim,num_layers=2, batch_first=True)
        self.hidden2label = nn.Linear(hidden_dim * 2, self.label_size)
    def forward(self, x):
        encodes = self.encoder(encodes, sum(lens))
        seq, start, end = [], 0, 0
        for i in range(self.batch_size):
            seq.append(self.get_zeros(max_len-lens[i])) # 标准化每个 ast 的语句长度
            seq.append(encodes[start:end])    start = end    end += lens[i]
        gru_out, hidden = self.bigru(encodes, self.hidden)
        gru_out = torch.transpose(gru_out, 1, 2)
        gru_out = F.max_pool1d(gru_out, gru_out.size(2)).squeeze(2) # pooling
        y = self.hidden2label(gru_out)
        return (gru_out , y)
if __name__ == '__main__':
    program_feature_model = ProgramFeatureModel()
    parameters = m.parameters()
    optimizer = torch.optim.Adamax(parameters)
    loss_function = torch.nn.CrossEntropyLoss()
    for batch_data in train_data:
        output, predict = program_feature_model(batch_train_features)
        loss = loss_function(predict, Variable(batch_train_labels))
        loss.backward()
        optimizer.step()
```

图 4-11: 警告特征表示关键代码

ProgramFeature 类的返回值为代码特征向量和神经网络预测结果。神经网络预测结果用于模型训练阶段，以不断优化网络参数；代码特征向量则应用于模型应用阶段，以提取代码特征向量。首先，调用语句编码器（Encoder）对每棵语句树进行编码，这样每棵语句树就转换成了数字向量；然后，由于不同警告代码拆分出来的语句树个数不等，所以需要采用填充对齐的方式统一语句的长度（sequence_len）。接着，得到了形如（batch_size , sequence_len , encoding_dim）格式的高维张量 torch 后，输入神经网路 GRU 中训练模型得到向量（batch_size , sequence_len , hidden_dim）。使用转置（transpose）、最大池化降维（max_pool1d）手段将向量转换成（batch_size ， hidden_dim）格式。这样，每个警告代码就被表示成定长为 hidden_dim 的向量了。最后，采用线性网络结构对特征向量进行计算，得到向量的预测标签值。

Main 函数中展示的是代码特征提取模型训练的简易过程。模型学习率设为 0.001，使用的损失函数是交叉熵损失函数，优化器是 Adam 优化器，训练周期设置为 30 次。训练结束得到的代码特征提取模型需存放在服务器中。

4.5 警告分类器模型构建模块实现

4.5.1 顺序图

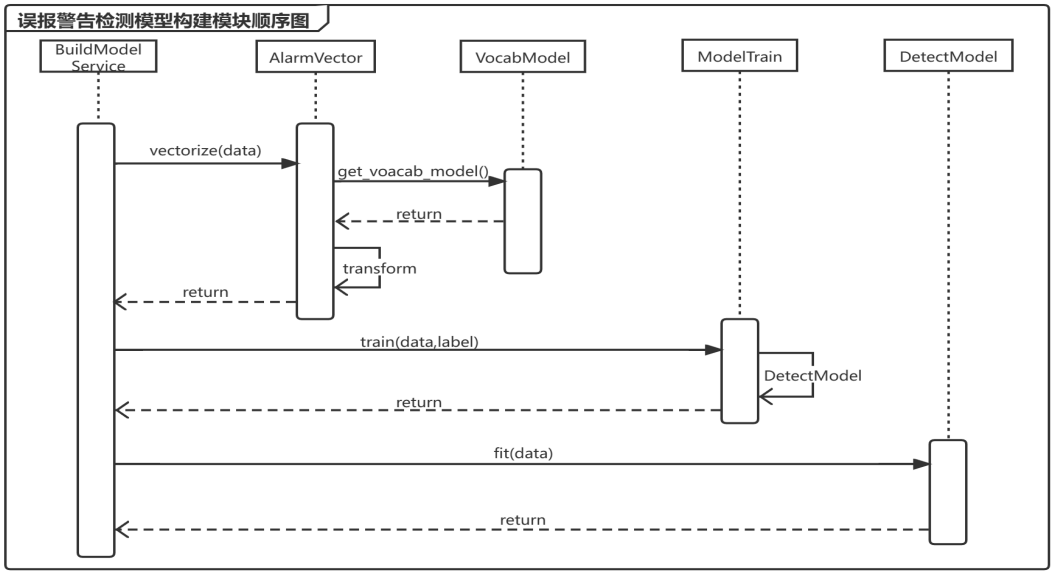


图 4-12: 警告分类器模型构建顺序图

如图4-12所示是警告分类器模型构建模块的顺序图。BuildModelService 类是本模块的入口，负责统一对外提供服务。BuildModelService 类首先会调用 AlarmVector 类的 vectorize 函数，用来获得警告的特征向量。AlarmVector 类会调用 VocabModel 类的 get_vocab_model 函数获得训练好的特征模型以提取警告特征；然后调用 ModelTrain 类的 train 函数在警告特征的基础上训练警告检测模型，并将该警告分类器持久化到服务器上；最后，调用 DetectModel 类的 fit 函数，可以得到待测警告的标签值。

4.5.2 关键代码

如图4-13所示为本系统中警告分类器模型构建模块的关键代码。该模型构建使用主流的机器学习框架 PyTorch，因此使用 Python 语言编写模型构建部分的代码。

```
class DetectModel(nn.Module):
    def __init__(self, hidden_dim, label_size):
        super(Linear, self).__init__()
        self.label_size = label_size
        self.hidden_dim = hidden_dim
        self.linear=nn.Sequential(nn.Linear(self.hidden_dim, self.label_size), nn.Dropout(p=0.2) ,
nn.Softmax(dim=None))
    def forward(self, x):
        f_len = len(x[0])
        x = np.array(x)
        x.astype(float)
        x = torch.from_numpy(x)
        x = x.view(-1, f_len)
        x = x.to(torch.float32)
        y = self.linear(x)
        return y
if __name__ == '__main__':
    detect_model = DetectModel()
    parameters = detect_model.parameters()
    optimizer = torch.optim.Adamax(parameters)
    loss_function = torch.nn.CrossEntropyLoss()
    for batch_data in train_data:
        predict = detect_model(batch_train_features)
        loss = loss_function(predict, Variable(batch_train_labels))
        loss.backward()
        optimizer.step()
```

图 4-13: 警告分类器模型构建关键代码

该模型采用了三层线性结构，它们的隐藏层维度为 225，150 和 100，按照隐藏层为输入层 75% 的比例设置（警告特征向量的长度为 300）。系统使用的

损失函数是交叉熵损失函数，优化算法是自适应学习率优化算法 Adam。训练周期为 30 轮，模型学习率设为 0.001，批量处理数据量设为 64。整个训练的流程依照机器学习技术学习模型的基本步骤，通过多轮训练周期的学习，直至模型收敛即训练完成。DetectModel 类是警告分类器模型的构建类，该类封装了分类器模型的基本结构。调用 forward 方法，即可反馈测试数据的正误报概率值。在 main 函数中，通过不断循环多轮次学习模型，依据损失函数计算的梯度值，使用优化器迭代优化模型参数。

4.6 智能化源码警告分类系统的实现

图 4-14为创建检测任务的页面，首先需要上传待检测项目文件，格式为 zip 或 tar.gz 的压缩包形式；然后填写项目名；最后填写项目描述。上传项目的大小不得超过 500MB。检测创建成功后步骤条会跳转到已完成状态，检测任务页面也会随之刷新，所有输入框的填写信息重置。

图 4-14: 创建检测任务页面

图 4-15为检测任务列表页面，可以查看当前用户所有的检测任务的信息，包括任务名、任务描述、创建时间、检测项目和任务进度，其中任务进度分为已创建、扫描中、已扫描、检测中和已检测等状态。同时还支持按照创建时间和检测进度进行筛选，该功能能够帮助用户快速找到未扫描和未检测的任务。每个检测任务单项的最后一栏是操作栏，支持执行扫描、执行检测、查看初始警告列表和查看检测结果的功能。其中查看初始警告列表和查看检测结果的功能分别需要扫描和检测完成后方可操作，相应的扫描或者检测操作未完成前，设置按钮状态为无法操作，从根本上避免了用户的错误触发行为。

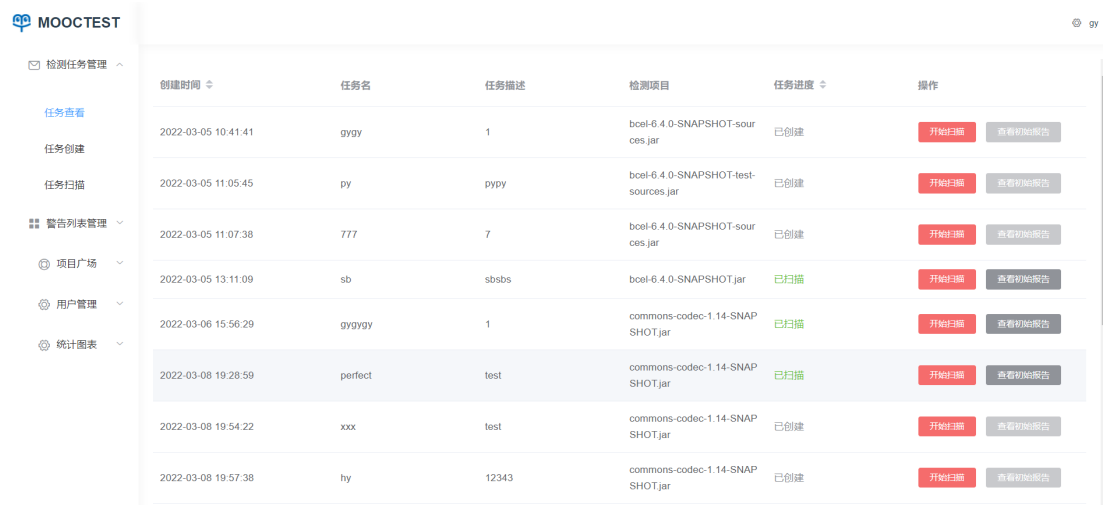


图 4-15: 检测任务列表页面

图4-16所示为用户某个检测任务的初始警告列表。该页面和检测任务列表页面很相似，包含警告 id、警告等级、警告文件、警告方法、警告方法、警告行、操作这几个数据项。其中操作栏只有详情查看一个按钮，点击详情查看按钮可以看到当前警告的详细信息。警告的详细信息会以面板的形式展示，关闭后会返回到初始警告列表页面。初始警告列表支持按照危险等级进行排序，默认情况下是降序排序。

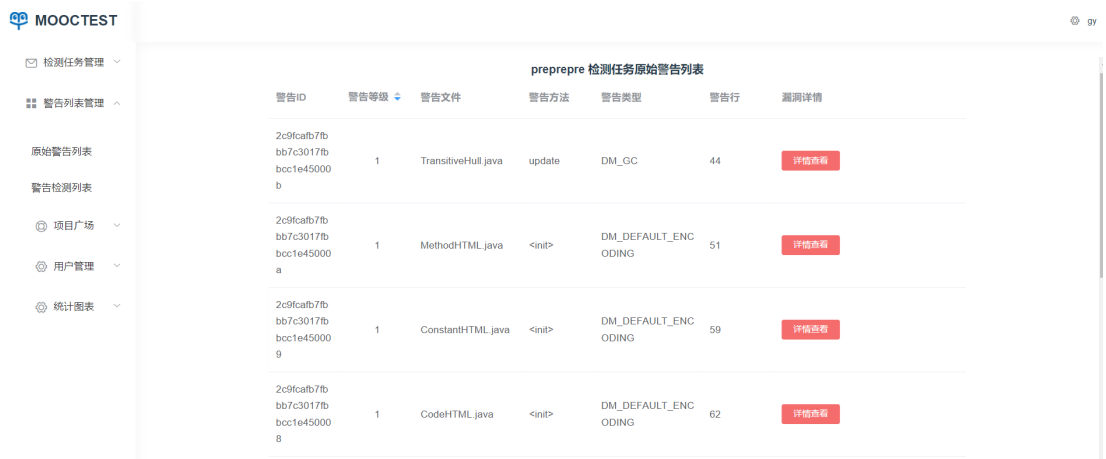


图 4-16: 原始警告列表页面

图4-17所示为警告检测结果页面。该页面展示了用户某个检测任务的检测结果。该页面主要有警告检测详情和警告按钮列表组成。警告检测详情部分，用户可以看到警告的具体信息，以及在警告检测过程中系统提取的警告特征，包括警告程序切片代码等等。警告检测详情部分的右侧是警告按钮列表，按钮

的配色有红和绿两种颜色，红色代表警告误报，绿色代表警告正报。每个按钮上都有一个数字，系统对每个警告进行了编号，代表着检测任务中的第几个警告。点击警告对应序号的按钮即可查看该警告的详情。警告详情下方展示了整个检测项目的正误报比例。

图 4-17: 警告检测结果页面

图4-18所示为检测任务统计信息的结果概述。该页面是由基本信息和统计信息两部分构成的，基本信息主要以描述列表的形式呈现，包括用户、任务名、总体警告数、检测文件名；统计信息主要通过图表的形式展现，包括根据1、2、3三个危险等级分类的警告数量绘制的条状图以及按警告类型进行分类的警告数量绘制的饼状图。在统计图中点击每个扇形区域，可以跳转到对应的警告列表页面。

图 4-18: 检测任务统计详情页面

4.7 本章小结

本章主要是对智能化源码警告分类系统的实现细节进行描述。按照模块划分，首先对警告数据集构建中使用的顺序图和关键代码进行详细描述；其次对代码预处理模块中的代码补全算法做了详细的讲解，并给出了整个模块的顺序图和标识符抽象的关键代码；随后借助顺序图和关键代码对代码特征提取模块进行说明，并给出了代码补全的具体算法；然后描述了警告特征表示模块中代码特征表示模型的结构，并讲解了关键代码；接着对警告分类器构建模块的模型结构作具体的阐释；最后对系统的主要界面以及相应的功能进行了介绍。

第五章　系统测试与实验分析

5.1　测试准备

5.1.1　测试目标

本系统的测试部分主要从三个方面着手进行，第一个方面是验证软件的性能和健壮性，主要关注于本系统在对各种项目进行某种操作时保持系统不崩溃，并且能够在规定时间内执行结束返回结果；第二个方面是验证系统的功能性是否完备，是否能够满足用户的需求；第三个方面是验证系统的性能是否比目前的方法效果更好。

根据第三章对智能化源码警告分类系统的需求分析、架构设计和第四章的系统具体实现，本章的系统测试与实验分析将包括以下几个部分：

健壮性和性能测试。该测试主要是针对系统在对各种项目的操作中，是否可以确保系统不崩溃且在规定时间内完成，从而保证系统的稳健性。

功能测试。该测试主要是检查系统是否可以提供相应的用户服务，且用户操作起来是否便捷，符合人机交互设计规范。

效果测试。该测试主要是检查系统的检测性能是否优于目前比较流行的误报检测模型，并且做了两个 RQ 来论证特征和模型之间的优劣。

5.1.2　测试环境

本系统部署在服务器上并进行测试，测试配置如表5-1所示。在硬件方面，由于误报警告检测是计算密集型任务，所以服务器是 16 核并且 CPU 选择了英特尔第六代微处理器架构 Intel Skylake，服务器内存为 32G。软件上有 Java、MySQL、Python 等。

表 5-1: 测试环境信息表

环境	基本信息
服务器	Ubuntu 16.04 LTS，32G 内存，16 Intel Core Processor (Skylake)
Java	jdk 1.8
Python	python 3.6.0
Mysql	mysql 8.0.0

5.2 健壮性和性能测试

本小节的测试目标是检测系统的健壮性和性能，即系统应对各种大小的项目输入，保持系统不崩溃以及在系统遇到崩溃的情况下，能够给予用户充分的反馈信息，以证明系统的稳健性。

该部分测试的主要思路是接受大量大小量级不一的项目，批量化对系统进行测试，观察系统是否能够正常运行，以及系统完成时间是否在系统设定的阈值以内。

具体步骤是首先收集大量的项目作为测试输入，然后监控系统在执行过程是否会存在异常状态，包括崩溃、卡死等现象，最后在系统完成时计算任务的执行时间，判定性能是否符合预期。下面介绍该部分测试使用的数据集，本文中使用的数据集主要来源于 Maven 中央仓库。本文尽可能选择 Apache 项目，根据特定的排序方式对 Maven 中央仓库中的项目进行排序，并使用 Git API 爬取 Top 30 的 Java 项目。

5.2.1 测试设计

表 5-2是扫描任务性能检测的测试用例。本文根据 Star 数量、近两年 commit 次数对 Maven 中央仓库的项目进行排序，选择 Maven 中央仓库排名 Top 30 的项目。在每个项目中选择使用次数最多的构建版本，构建版本需打包成 Jar 的形式，作为测试对象批量执行扫描操作进行测试，对于每一个项目，本文设置系统的扫描时间阈值为半小时，如果扫描执行时间超过半小时，则认为系统扫描耗时太长，超时项目数量越多，系统性能越低。

表 5-2: 扫描任务性能检测

测试 ID	TC1
测试接口	源代码警告扫描接口
测试方案	1. 筛选并下载 Top30 的 Maven 项目； 2. 选择每个项目使用最多的构建版本（Jar 包）作为测试对象； 3. 批量化执行扫描操作。

表 5-3是检测任务性能检测的测试用例。该测试用例检测的是系统警告分类器模型检测性能的可靠性。本文将扫描任务性能检测完成后得到的初始警告列表，作为检测任务性能检测的输入。本文以每个项目中的初始警告列表看作一个整体，执行批量化警告检测操作。对于每一个项目，本文设置系统的检测时间阈值为 1 小时，如果检测执行时间超过 1 小时，则认为系统检测耗时太长，超时项目数量越多，系统性能越低。

表 5-3: 检测任务性能检测

测试 ID	TC2
测试接口	警告检测接口
测试方案	1. Top30 的 Maven 项目扫描完成后得到的初始警告列表作为测试对象； 2. 接收初始警告列表作为输入，批量化执行警告检测操作。

5.2.2 测试执行

表 5-4列出了扫描任务的测试结果。总共收集的 30 个项目，其中有 2 个构建形式为 aar 格式，由于系统只能输入 Jar 包格式，因此它们无法执行测试，实际参与测试的项目只有 28 个。在 28 个项目中，有 8 个 Jar 文件内无“.class”文件，由于系统具体扫描的是一个个二进制’.class’ 文件，而 Jar 包内无可扫描的文件，系统会在执行扫描时检测并提示用户 Jar 包不合法，并正常退出。对于剩下的 20 个项目，系统能在最长扫描时间内完成扫描，即本文认为扫描成功率为 100%，该系统扫描任务具有较高的健壮性。

在扫描任务性能方面，可以看出系统最短扫描时间仅有 8.31s，对应的 Jar 包为 commons-bcel-6.4.0-SNAPSHOT.jar，这一类 Jar 包的特点在于本身项

表 5-4: 扫描任务健壮性和性能测试结果

测试 Jar 包	合法 Jar 包	扫描成功率	最短扫描时间 (s)	平均扫描时间 (s)	最长扫描时间 (s)
30	20	100%	8.31	45.22	989.21

目规模较小，项目中能够扫描的二进制文件也相对较少，因此其扫描时间也相应较短；系统平均扫描时间为 45.22s，这说明对于绝大多数的项目，用户可以在一分钟以内完成扫描；系统最长扫描时间为 989.21s，对应的 Jar 包为 camel-core-2.21.5.jar，该 Jar 包大小为 4685KB，项目规模算是比较大的了，扫描完成得到了 1886 个警告，因此扫描时间很长，但是本系统仍能在半小时之内完成扫描，对于像 camel-core 这样的大型项目而言，本文认为这是能够被用户接受的。

表 5-5列出了检测任务的测试结果。对于扫描任务获得的 20 个项目的初始警告列表，系统能在最长检测时间内完成每一个项目的警告检测，即本文认为检测成功率为 100%，该系统检测任务具有较高的健壮性。 在检测任务性能

表 5-5: 检测任务健壮性和性能测试结果

最少警告数	平均警告数	最多警告数	最短检测时间 (s)	平均检测时间 (s)	最长检测时间 (s)	检测成功率
10	389.4	1456	11.23	402.71	1500.64	100%

方面，可以看出系统最短检测时间仅有 11.23s，对应的 Jar 包为 commons-bcel-6.4.0-SNAPSHOT.jar，该 Jar 包恰恰就是最少警告数的项目；系统平均检测时间为 402.71s，这说明对于绝大多数的项目，用户可以在十分钟以内完成检测；系统最长检测时间为 1500.64s，对应的 Jar 包为 camel-core-2.21.5.jar，该 Jar 包也是做多警告数的项目，因此检测时间很长，但是本系统仍能在半小时之内完成扫描，对于像 camel-core 这样的大型项目而言，本文认为这是能够被用户接受的。从测试结果可以发现，检测时间和警告数量是基本成正比的，平均下来每个警告检测时间为一秒多。而警告检测最耗时的部分在于特征提取，这足以证明经过本系统代码预处理后的警告代码在去除了无效噪音后，不会因警告类或者警告方法的规模太大而导致警告代码过多，从而保证大部分的警告能在很短的时间内将特征提取出来并且该特征更能表征警告代码。

综上，系统的扫描耗时和检测耗时是基本成正相关的。扫描耗时越长，则意味着项目复杂度高，产生的警告数也随之上升，警告数的上升直接影响了检测时间的上升。本系统对于各类 Jar 包具体有较高的健壮性，并且能较为及时的执行完成扫描任务和检测任务，大型项目的扫描时长和检测时长在用户的可接受范围之内。

5.3 功能测试

本小节测试的目的是为了测试智能化源码警告分类系统的功能可以满足系统用户的预期。根据第三章功能性需求中的系统用例来设计测试用例，按照系统对外提供的服务进行测试，以达到功能覆盖的标准，保证最终的系统符合用户的实际诉求，基本满足支撑误报警告智能化检测系统的业务需求。

5.3.1 测试设计

表 5-6展示了查看检测任务列表的详细测试步骤预期结果，该测试用例是根据用例描述 UC1 进行设计的，测试能否全面覆盖查看检测任务列表功能。

表 5-6: 查看检测任务列表测试用例

测试 ID	TC3
测试名称	查看检测任务列表
测试功能	系统用户可以查看自己的检测任务列表、使用关键字搜索（模糊）查询任务、根据各个数据项排序检测任务列表，筛选已创建、已扫描和已检测的检测任务。
测试步骤	1. 系统用户登录系统； 2. 筛选已创建状态的检测任务； 3. 筛选已扫描的检测任务； 4. 筛选已检测的检测任务； 5. 搜索框输入模糊查询条件； 6. 点击搜索按钮。
预期结果	1. 显示该用户所有的检测任务； 2. 显示该用户已创建的检测任务； 3. 显示该用户已扫描的检测任务； 4. 显示该用户已检测的检测任务； 5. 显示模糊条件相关的检测任务。

表5-7展示触发警告扫描的详细测试步骤和相应的预期结果，测试的关注点是系统是否能够正确的被用户触发源代码警告扫描功能，以及扫描过程中任务状态的变化是否正确和扫描完成后用户是否能够查看初始警告列表。

表 5-7: 触发警告扫描测试用例

测试 ID	TC4
测试名称	触发警告扫描
测试功能	系统用户可以触发警告扫描流程，扫描完成后“查看报告”按钮变为可点击状态。
测试步骤	1. 选择一个任务状态为“已创建”的检测任务； 2. 点击“开始扫描”按钮，检测任务状态是否转变成“扫描中”状态； 3. 扫描完成后，“查看报告”按钮是否从不可操作状态变为可点击状态； 4. 点击“查看报告”按钮是否会显示初始警告列表。
预期结果	1. 显示状态为“已创建”的检测任务列表； 2. 检测任务状态由“已创建”转变为“扫描中”； 3. “查看报告”按钮从不可操作状态变为可点击状态； 4. 显示初始警告列表页面。

表 5-8: 触发警告检测测试用例

测试 ID	TC5
测试名称	触发警告检测
测试功能	系统用户可以触发警告检测，系统会应用内置的误报检测模型筛选虚假警告，最终获得真实警告列表。
测试步骤	1. 选择一个任务状态为“已扫描”的检测任务； 2. 点击“开始检测”按钮，检测任务状态是否转变成“检测中”状态； 3. 检测完成后，“查看检测结果”按钮是否从不可操作状态变为可点击状态； 4. 点击“查看检测结果”按钮是否会显示真实警告列表。
预期结果	1. 显示状态为“已扫描”的检测任务列表； 2. 检测任务状态由“已扫描”转变为“检测中”； 3. “查看检测结果”按钮从不可操作状态变为可点击状态； 4. 显示真实警告列表页面。

表5-8展示了触发警告检测的详细测试步骤和相应的预期结果，测试系统内置的警告分类器模型检测初始警告列表中误报警告的能力，主要关注点是警告分类器模型的应用是否可以顺利使用，以及检测过程中的一些状态改变是否正确。

表 5-9: 查看警告详细信息测试用例

测试 ID	TC6
测试名称	查看警告详细信息
测试功能	系统用户可以查看警告详细信息，包括警告名称、警告类型、风险等级等报告信息和警告切片、警告度量值等系统提取信息。
测试步骤	1. 用户点击“查看检测结果”按钮，是否跳转到真实警告列表页面； 2. 点击“警告详情”按钮，是否显示警告详细信息； 3. 点击“结果概述”按钮，是否展示检测结果的总览信息； 4. 点击警告等级和警告类型饼状图，是否显示相应的警告列表。
预期结果	1. 系统跳转到真实警告列表页面； 2. 系统显示警告的详细信息； 3. 系统展示检测结果的总览信息； 4. 系统显示相应的警告列表。

表5-9展示了查看警告详细信息的具体测试步骤和相应的预期结果，测试的是查看警告详细信息的各项细节，主要关注点是警告详细信息是否完备，以及总体概述信息和单体警告信息之间的跳转是否符合人机交互设计规范。

5.3.2 测试执行

本人严格按照上面设计的测试步骤执行测试用例，并将系统使用的实际结果与测试用例的预期结果进行比较。功能测试的结果记录在表5-10中。从表中可以看出，实际结果和预期结果基本一致，完全满足了系统的功能性需求，整个系统有着很高的可用性。

表 5-10: 功能测试用例执行结果

测试用例 ID	对应用例描述	测试结果
TC3	UC1	通过
TC4	UC2	通过
TC5	UC3	通过
TC6	UC4	通过

5.4 效果测试

本小节测试的目的是检测系统使用的警告分类器模型是否可以检测到误报警告，本系统检测误报警告的效果相比于目前主流的其他方法是否更加有效，以及哪个模型和哪种特征在警告分类中表现的最好。

相比于传统的警告扫描系统，本系统主要使用先进的特征表示方法并基于大型的真实警告数据集上，构建了性能表现优秀的警告分类器模型，从而提升了误报检测的准确性。在本小节，将从数据集、参数设置和评估度量三个方面描述测试的配置信息并通过三组 RQ 来证明本系统相较于目前解决方案表现出的巨大优势。

5.4.1 数据集描述

表 5-11: 数据集信息表

项目名称	版本数	平均警告数	平均正报数	平均误报数
camel-core	5	1870	208	1662
hadoop-common	4	930	100	830
lucene-core	18	507	88	419
solr-core	18	1448	264	1184
cassandra-all	11	1012	47	965

本小节使用的数据集如表 5-11所示。由于数据量的大小直接影响机器学习的性能，在过小的数据集上做机器学习往往容易过拟合，而本小节的实验是以项目的前一个版本的警告作为训练集，后一个版本的警告作为测试集，因此

对于项目中的每一个版本产生的警告数量具有一定的约束要求。本文在根据 Star 数目、近两年 Commit 次数筛选项目的前提下，要求项目的每个版本产生警告数量大于 500，方可作为效果测试的测试对象。最终符合条件的有如下五个项目：camel-core、hadoop-common、lucene-core、solr-core 和 cassandra-all。为了防止本文设定的 500 阈值过小导致过拟合，本小节使用的项目数据量分布在 500、1000、1500、2000 上下的都有，避免后续论证可能会带来的论据不充分性。

图5-1展示了一个警告实例。该警告来自于源码静态扫描工具 FindBugs 得到的初始警告报告，报告是以 xml 文件的形式存储的。在报告文件中，BugInstance 节点是警告实例的根节点，包含 Class、Method 和 SourceLine 三种类别的子节点，分别代表警告所属类、警告所属函数和警告所在行，节点内记录了警告相关的信息。其中 Class 节点只有一个，由于函数内可能存在代码调用关系，一个警告的 Method 和 SourceLine 节点往往存在多个。我们默认取警告的第一个 Method 节点和 SourceLine 节点作为其直接属性节点。后续在警告源码的基础上，对警告报告提取的警告信息做进一步的静态分析处理，可以获取到警告中代码调用关系的相关信息。

```
<BugInstance type="DM_DEFAULT_ENCODING" priority="1" rank="19" abbrev="Dm" category="I18N" instanceHash="8c66cd250a37837e3497e9f5107dbcb6" in
  <ShortMessage>Reliance on default encoding</ShortMessage>
  <LongMessage>Found reliance on default encoding in org.apache.camel.BytesSource.getReader(): new java.io.InputStreamReader(InputStream)</LongMessage>
  <Class classname="org.apache.camel.BytesSource" primary="true">
    <SourceLine classname="org.apache.camel.BytesSource" start="38" end="63" sourcefile="BytesSource.java" sourcepath="org/apache/camel/BytesSource.java"
      <Message>At BytesSource.java:[lines 38-63]</Message>
    </SourceLine>
    <Message>In class org.apache.camel.BytesSource</Message>
  </Class>
  <Method classname="org.apache.camel.BytesSource" name="getReader" signature="()Ljava/io/Reader;" isStatic="false" primary="true">
    <SourceLine classname="org.apache.camel.BytesSource" start="55" end="55" startBytecode="0" endBytecode="53" sourcefile="BytesSource.java" sourcepath=
    <Message>In method org.apache.camel.BytesSource.getReader()</Message>
  </Method>
  <Method classname="java.io.InputStreamReader" name="&lt;init&gt;" signature="(Ljava/io/InputStream;)V" isStatic="false" role="METHOD_CALLED">
    <SourceLine classname="java.io.InputStreamReader" start="72" end="79" startBytecode="0" endBytecode="108" sourcefile="InputStreamReader.java" sourcep
    <Message>Called method new java.io.InputStreamReader(InputStream)</Message>
  </Method>
  <SourceLine classname="org.apache.camel.BytesSource" primary="true" start="55" end="55" startBytecode="8" endBytecode="8" sourcefile="BytesSource.java
    <Message>At BytesSource.java:[line 55]</Message>
  </SourceLine>
</BugInstance>
```

图 5-1: 警告实例

5.4.2 评估度量

本文将本系统的警告分类器模型与目前流行的几种分类模型做对比。将项目前一个版本的警告数据作为训练集，后一个版本的警告数据作为测试集，以准确率（Accuracy）、召回率（Recall）、精确率（Precision）和 F1 作为度量比较警告分类器的效果。为了解释这些度量指标的计算方法，首先介绍如表5-12所示的分类问题混淆矩阵。对于每一个警告，警告分类器会对其报告为

表 5-12: 分类问题混淆矩阵

		检测结果	
		警告为真	警告为假
真实结果	警告为真	TP	FN
	警告为假	FP	TN

真警告和假警告，因此产生实际为真警告且检测结果为真警告的数 TP，实际为真警告而检测结果为假警告的用例数 FN，实际假警告而检测结果为真警告的用例数 FP 和实际假警告且检测结果为假警告的用例数 TN，在此基础上，警告分类器的准确率计算方式为 $Accuracy = \frac{TP + TN}{TP + FN + FP + FN}$、精确率计算方式为 $Precision = \frac{TP}{TP + FP}$、召回率计算方式为 $Recall = \frac{TP}{TP + FN}$ 以及 F1 计算方式为 $F1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$，取所有版本的平均值作为每个警告分类器的最终结果。

5.4.3 实验设置

我们依据研究现状，将本文的方法与目前最先进的三种警告分类方法做了对比，所有方法的相关信息如表 5-13所示。

BaseLine1 来源于 Koc[4]，该文首先提取了警告代码的程序切片并将其表现为字节码的形式；接着在警告字节码上采用删除表达式和替换程序特定 token（特定对象替换成”UNK_OBJ”，函数调用替换成”UNK_CALL”）两种策略，以达到弱化噪音 token 对警告分类效果的影响；最后使用 Word2Vec 对 token 进行编码并应用 LSTM 来发现与误报警告相关的代码结构。

BaseLine2 是 Lee 与三星合作的一篇研究 [5]，该研究是在三星持续集成系统提供的数据上，构建了一个基于卷积神经网络（CNN）的分类器。该方法不需要任何特征工程或提取，只需将警告上下五行代码 token 化即可。分类器使用 Word2Vec 的 skip-gram 模型对警告上下五行代码的 token 进行编码，将语料库中的 token 嵌入到向量中。最终在 token 级别的警告上下五行代码上学习与误报警告相关的词汇模式。

BaseLine3 和 BaseLine1 的方法源自同一个作者 Koc[7]。该文在警告程序切片上提取警告抽象语法树并采取先根遍历的方式获得带有语法结构的 token 序列，然后在 token 序列上使用词袋模型的方式对警告做特征向量化处理，最后在提取的警告特征上运用 LSTM 学习警告分类器。

由于 BaseLine1 在警告字节码上已经去除程序特有属性，因此除了 BaseLine1 外，我们使用本文代码预处理部分的标识符抽象方法，对 BaseLine2 和 BaseLine3 的实现做了一些改进。我们抽象化了警告代码中的关键词、变量名称、常量值等，减少了警告 token 的数量，泛化了 Word2Vec 训练得到的语料库，一定程度上能够提高模型（相比较原文）的分类效果。对于这三个 BaseLine 的其余参数设置，本文遵循其原始论文或者发布代码中的描述。

本系统的方法使用 javalang 提取警告程序切片的抽象语法树，使用 Word2Vec 的 skip-gram 模型 [64] 训练语法树的符号嵌入，并将嵌入大小设置为 128。由于编码语句树的向量维度是 128，根据隐藏层一般设为输入层节点数的 75%，我们将代码特征向量提取模型的隐藏层维度设为 100。系统使用静态分析工具 SourceMonitor 提取警告手工特征，并从警告报告中分离与警告相关的统计特征，两者结合构成手工特征，得到的所有手工特征如表 5-15所示。警告分类器模型中三层线性结构的隐藏层维度为 225，150 和 100，同样按照 75% 的比例设置（本文方法提取的警告特征向量的长度为 300），使用 SoftMax 激活函数使分类结果非线性化。本系统将批量数据处理的大小设置为 64，训练轮次设置为最多 30 个 epoch。系统使用学习率为 0.001 的优化器 AdaMax[65] 进行训练。所有实验均在具有 16 核 2.4GHz CPU 的服务器上进行。

表 5-13: BaseLine 实验内容

BaseLine 编号	特征来源	特征向量表示	模型
BaseLine1	字节码	Word2Vec	神经网络 LSTM
BaseLine2	上下五行代码	Word2Vec	神经网络 CNN
BaseLine3	程序切片	Bow	神经网络 LSTM
本文方法	程序切片	代码特征 + 手工特征	神经网络 GRU

5.4.4 有效性

本文的方法是否能够超过已有的源码警告检测方法？为了验证本文方法的有效性，本文比较了 BaseLine 方法的分类效果。所有方法的实验结果如

表5-14所示。可以无论是在哪个项目上，本文的方法在F1值上都远远超过了三组BaseLine，这足以证明本文提出的方法能够提升分类器的分类效果。本文方法的Recall值和Precision值同样遥遥领先于前三种方法。分类器检测的警告保持最高的正报率和召回率，因此本系统提出的方法是真实有效的。

表 5-14: BaseLine 对比实验结果

BaseLine 编号	项目名	**Precision**	**Recall**	**F1**
BaseLine1	camel-core	0.7025	0.7836	0.7409
	cassandra-all	0.9184	0.8862	0.9020
	hadoop-common	0.9545	0.2210	0.3589
	lucene-core	0.8039	0.3059	0.4432
	solr-core	0.4249	0.7619	0.5456
BaseLine2	camel-core	0.6945	0.9182	0.7908
	cassandra-all	0.9599	0.8606	0.9075
	hadoop-common	0.3546	0.2561	0.2921
	lucene-core	0.9801	0.0612	0.1153
	solr-core	0.3874	0.6915	0.4966
BaseLine3	camel-core	0.7988	0.6617	0.7238
	cassandra-all	0.9314	0.8974	0.9141
	hadoop-common	0.3546	0.2561	0.2921
	lucene-core	0.8823	0.2307	0.3658
	solr-core	0.8841	0.3606	0.5123
本文方法	camel-core	**0.9281**	**0.9021**	**0.9140**
	cassandra-all	0.9382	0.8985	0.9179
	hadoop-common	0.8000	0.6315	0.7058
	lucene-core	0.7662	0.6344	0.6941
	solr-core	**0.9891**	0.8888	**0.9411**

本文方法分类效果之所以如此突出，主要由以下两点造成的：其一，与NLP中的长文本类似，在提取抽象语法树的代码特征过程中，尤其是当树非常大且很深时，因为产生的token数量过于庞大，容易出现梯度消失问题，即在训练过程中梯度变得非常小。因此，以自下而上的方式或使用滑动窗口技术遍历和编码整棵语法树时可能会丢失长期上下文信息。本文中的三个BaseLine都

是首先对警告相关的 token 进行编码，然后再应用高级神经网络学习 token 之间的联系。本文的方法是先将整棵语法树进行拆分，然后分别对拆分得到的每棵语句树进行编码，最后再使用机器学习技术学习语句之间的顺序关联。由于语句级别的特征向量粒度更粗，编码得到的向量数量要远远少于前三种方法，因此这么做不仅能够解决传统的 token 编码方法在代码特征提取过程中出现的长期依赖问题，而且相较于 token 之间的联系，学习语句间的关联性更能发现警告的正报模式。其二，本文方法在代码特征的基础上，融合了手工特征，能够挖掘出更加丰富的警告特性。手工特征列表如表 5-15所示，涵盖了警告切片、警告类、警告函数和警告报告四个方面的警告信息。本文中的三个 BaseLine 仅仅提取了警告代码相关的特征，特征维度较为单调，得到的警告特征所能表现警告的信息也就相对匮乏。

5.4.5 实用性

经过多项研究表明，程序切片可以将代码片段缩减成最小形式，有效减少无关代码噪音，能够最大限度地保留代码的上下文语义特征。因此，在基于程序切片作为特征来源的前提下，本文采取控制变量的方式探究特征向量表示方式和模型使用对分类效果的影响。

为了充分验证传统模型对分类效果的影响，泛化传统模型的一般性，本文在传统模型的选择中使用了决策树、随机森林和 KNN 这三种传统模型。与此同时，本文选择了融合特征，代码特征和度量特征作为不同的特征输入角度，采用交叉组合的方式产生了十二组实验。其中，融合特征是是本文方法所使用的警告特征，通过将代码特征和手工特征扁平化处理后得到。总体的实验结果如表 5-16所示。

融合特征是否更能表现警告的特征？可以看出在使用相同模型的情况下，融合特征的 F1 值基本上都要好于代码特征和手工特征。这足以证明融合特征结合了警告在代码特征和手工特征所表现出来的特性。从数据中我们还能发现融合特征比代码特征的分类效果要好，但是提升幅度不大，这说明手工特征能够捕获到代码特征以外的警告特性，但代码特征相比手工特征更能学习到警告的误报模式，很多误报警告都能从警告代码的语法结构中辨别出。

表 5-15: 手工特征列表

特征类别	特征含义	特征编号
Slice	警告切片结果的代码行数	1
	警告切片结果的条件分支数量	2
	警告切片结果的警告数量	3
	警告切片结果的圈复杂度	4
Function	警告所在函数的代码行数	5
	警告所在函数的条件分支数量	6
	警告所在函数的圈复杂度	7
	警告所在函数的深度	8
	警告所在函数的 caller 次数	9
	警告所在函数的 callee 次数	10
	警告所在函数中的警告数量	11
File	警告所在文件的代码行数	12
	警告所在文件的条件分支数量	13
	警告所在文件的函数的个数	14
	警告所在文件的深度	15
	警告文件的圈复杂度	16
	警告所在文件的警告数量	17
Warning report	警告 type	18
	警告 category	19
	警告危险等级	20
	警告排名	21
	警告报告中的代码行数	22

神经网络是否比传统机器学习模型分类效果更好？在使用相同特征的情况下，神经网络学习模型的 F1 值要更高。这说明了神经网络学习模型在警告分类的预测上效果更好。这是由于面对大量的警告数据，神经网络在训练过程中以可解释性为代价，通过不断地学习获得警告的误报模式，并且其在图像和自然语言处理中本身就具有很大的优势。与此同时，在传统机器学习模型中，随机森林分类器的 F1 值又要远远高于其他两个，这说明随机森林在机器学习模型中是用于警告分类的最佳模型。由于随机森林综合多个决策给出分类结果，

因此效果较好就不难理解了。

表 5-16: RQ 实验结果

实验编号	使用特征	使用模型	Precision	Recall	F1
实验 1	融合特征	GRU	**0.9281**	**0.9021**	**0.9140**
实验 2	手工特征	GRU	0.0994	0.8684	0.1784
实验 3	代码特征	GRU	0.8233	0.9415	0.8784
实验 4	融合特征	决策树	0.7911	0.9468	0.8619
实验 5	手工特征	决策树	0.9291	0.5462	0.6880
实验 6	代码特征	决策树	0.7911	0.9468	0.8619
实验 7	融合特征	KNN	0.9003	0.8712	0.8851
实验 8	手工特征	KNN	0.8780	0.7058	0.7826
实验 9	代码特征	KNN	0.8815	0.8724	0.8761
实验 10	融合特征	随机森林	0.8997	0.8785	0.8889
实验 11	手工特征	随机森林	0.8732	0.8732	0.8732
实验 12	代码特征	随机森林	0.8813	0.8746	0.8777

5.5 存在问题

在实验过程中，因为是以项目的前一个版本的警告作为训练集，后一个版本的警告作为测试集，每个版本的警告构成了一个小的警告数据集，项目各版本间的警告互不相干，因此可以规避多版本警告数据的重复性问题。

然而系统在警告数据集构建过程中，由于 SpotBugs 扫描的多版本警告都包含在数据集中，因此引发了如下两个问题。其一，也是最严重的**警告数据重复性问题**。每一次 commit 并不会修复所有的代码缺陷，因此相同警告对可能会出现在多个版本中。假设项目 p 有 n 个版本 $v_1, v_2, \ldots, v_i, \ldots, v_n$，考虑到一种最坏的情况，假设警告在项目的 v_1 到 v_{n-1} 版本中都出现，在 v_n 中被修复，那么相同的警告将出现在 $v_1, v_2, \ldots, v_{n-1}$ 中，并带有相同的误报标签，即出现 $n-1$ 个重复警告，造成数据的大量冗余。其二，**警告标签前后不一致问题**。该问题主要是由 SpotBugs 工具产生误报造成的，出现的概率较小。假设警告在 v_1 中出现，v_{n-2} 中消失，v_{n-1} 和 v_n 中又出现，那么根据差分分析的警告标记策略，警告在 v_1 中是正报，在 v_{n-1} 中是误报，从而导致了相同警告的标签不一致问题。

5.6 本章小结

本章主要是对智能化源码警告分类系统的功能性需求和非功能性需求进行全面的测试。功能性需求的测试是针对第三章系统用例描述的功能进行逐个测试，验证系统是否满足了用户的实际需求，提供的功能是否完备。非功能性需求的测试主要分为健壮性、性能测试和效果测试。健壮性和性能测试主要是对系统扫描任务和检测任务的稳健性进行测试。效果测试主要是对系统中构建的警告分类器的效果进行测试，验证了本文提出方法构建的警告分类器模型确实优于目前几种主流的做法，并且讨论了模型和特征对分类模型构建的影响。

第六章　总结与展望

6.1　总结

源码静态扫描工具因为能够快速便捷地检测出项目中的代码缺陷而受到开发者的广泛使用。由于源码静态扫描工具是在编译时执行扫描的，因此往往过度估计了程序的执行路径，导致产生了大量的误报警告。为了找到正报，开发者需要花费大量时间去手动检测警告。为了减轻开发者审计的压力，现有的方法常常采用机器学习技术来检测正报。该方法是在一组警告的手工特征或者代码特征上，应用机器学习来学习警告分类器。然而这类方法还存在一些不足之处：其一，目前的研究大都采用合成测试数据集。该类型数据集数据量很大，但是其注入的代码缺陷都是人为刻意引入的，不能代表真实世界的项目，不具有泛化性。因此警告分类器的训练缺乏大量的真实警告数据集；其二，很多方法把警告代码视为自然语言文本来学习分类模型，将文本处理成 token 序列或者 token 词袋，这会遗漏代码的重要语义和语法信息。部分方法使用手工特征学习模型，尽管提取了警告某个方面的特征，但是缺少警告代码中包含的语法特征，并不能全面地表示一个警告。因此警告在特征表示方面还缺少一个全面的警告特征。

本文的主要工作：构建了一个大规模，真实项目的警告数据集和提出了一种全面的融合性代码特征。我们使用了一种基于差分分析的标记方法，用于构建一个大规模，真实项目的警告数据集。首先，选取 Apache 旗下高 star、多 commit 的项目，并下载近两年的所有版本；接着，结合三种警告比对策略（基于位置的比对策略，基于文本的比对策略，基于哈希的比对策略），版本间两两比对统计警告的出现情况；最后，采用差分分析法对警告进行正误报标记。

本文提取了一种全面的融合性代码特征。首先，使用程序切片、抽象语法树等静态分析技术对警告源代码做预处理；接着，构建基于抽象语法树的神经网络来学习源代码片段的向量表示，以捕获语句的词汇、语句级句法知识的自然性。该模型将大的警告代码抽象语法树分解成小语句树的序列，通过递归编码多路语句树得到语句向量，最后利用语句的自然性学习代码片段的向量表

示；然后，基于专家给出的意见，定义并提取了圈复杂度、警告危险等级、警告内函数调用数等 10 个手工定义特征；最后，使用扁平化技术融合代码片段向量表示和手工定义特征，并应用机器学习训练误报警告分类器模型。

6.2 进一步工作展望

智能化源码警告分类系统旨在对警告报告进行检测，该系统还有诸多可优化之处：

收集更多的警告数据集。由于本系统使用差分分析法来标记警告，差分分析法的标记思路是基于项目多版本警告列表的，而同一警告对可能会出现在多个版本中，因此数据集中会出现部分冗余警告，这对于模型的训练来说会产生很大的影响。然而去除冗余警告，又会导致数据集量级过小，模型训练效果不佳。后续可以考虑多收集几个大型项目的警告。

提升警告代码预处理的成功率。本系统在代码预处理阶段做了很多工作，包括程序切片处理和抽象语法树修改（代码补全和标识符抽象），程序切片借助了 Joana 工具，语法树修改使用了 JavaParser 工具。这两种预处理手段都属于静态分析范畴，由于语言在不断地发展抑或是工具本身所存在的局限性，静态分析并不能保证所有的警告都能处理成功。后续可以从工具本身考虑，使用更强大的静态分析工具或者综合使用多个静态分析工具。

选择更好的优化函数。本系统在代码特征模型构建和警告分类器模型构建部分采用的优化函数都是 Adam 函数，选择比较随意，没有实验数据支撑，只是为了优化模型参数。后续可以使用对比试验的方式找出最佳的优化函数。

致　　谢

论文写到这里即将进入尾声。我首先想感谢我的导师陈振宇老师和房春荣老师在研究生期间对我的谆谆教导。在毕设开题前，感谢两位导师耐心地为我指导，确定了毕设的总体研究方向。在系统开发阶段，感谢两位老师采取每月阶段性汇报的方式了解我的开发进度，致使我能把握好节奏，不紧不慢地完成系统开发。在论文写作阶段，感谢两位老师对论文进展的督促和关注。

我还要感谢葛修婷学姐。葛修婷学姐在整个毕设的设计中给我提供了很大的帮助，从系统的总体设计思路到后面的实验分析，都给我提出了很多宝贵的建议。同时，我也要感谢钱美缘同学，在科研的道路上互相鼓励，交流心得，使科研之路变得不那么枯燥无味，反而更加充满了动力。

我还要感谢我的父母。谢谢你们在每周都会和电话沟通，及时了解我的学习状况。在我遇到困难的时候给予了最大的帮助，保证我能按时完成毕设。

最后，我要感谢我的大学。谢谢南京大学软件学院为了提供一个教学质量高超、学术氛围浓厚的学习环境，让我在六年的大学生涯中学有所成，也因此找到了心仪的工作。

参考文献

[1] WANG S, LIU T, TAN L. Automatically learning semantic features for defect prediction[C] // 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). 2016 : 297 – 308.

[2] 李舟军, 张俊贤, 廖湘科, et al. 软件安全漏洞检测技术 [J]. 计算机学报, 2015, 38(4) : 717 – 732.

[3] 樊晓光, 褚文奎, 张凤鸣. 软件安全性研究综述 [D]. [S.l.] : [s.n.], 2011.

[4] KOC U, SAADATPANAH P, FOSTER J S, et al. Learning a classifier for false positive error reports emitted by static code analysis tools[C] // Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. 2017 : 35 – 42.

[5] LEE S, HONG S, YI J, et al. Classifying false positive static checker alarms in continuous integration using convolutional neural networks[C] // 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST). 2019 : 391 – 401.

[6] 张林, 曾庆凯. 软件安全漏洞的静态检测技术 [J]. 计算机工程, 2008, 34(12) : 157 – 159.

[7] KOC U, WEI S, FOSTER J S, et al. An empirical assessment of machine learning approaches for triaging reports of a java static analysis tool[C] // 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST). 2019 : 288 – 299.

[8] BISSYANDÉ T F, THUNG F, WANG S, et al. Empirical evaluation of bug linking[C] // 2013 17th European Conference on Software Maintenance and Reengineering. 2013 : 89 – 98.

[9] NGO K-T, DO D-T, NGUYEN T-T, et al. Ranking Warnings of Static Analysis Tools Using Representation Learning[J]. arXiv preprint arXiv:2110.03296, 2021.

[10] AVGUSTINOV P, BAARS A I, HENRIKSEN A S, et al. Tracking static analysis violations over time to capture developer characteristics[C] // 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering : Vol 1. 2015 : 437 – 447.

[11] ZHANG J, WANG X, ZHANG H, et al. A novel neural source code representation based on abstract syntax tree[C] // 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). 2019 : 783 – 794.

[12] MUSKE T, SEREBRENIK A. Survey of approaches for handling static analysis alarms[C] // 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM). 2016 : 157 – 166.

[13] LEE W, LEE W, YI K. Sound non-statistical clustering of static analysis alarms[C] // International Workshop on Verification, Model Checking, and Abstract Interpretation. 2012 : 299 – 314.

[14] ZHANG D, JIN D, GONG Y, et al. Diagnosis-oriented alarm correlations[C] // 2013 20th Asia-Pacific Software Engineering Conference (APSEC) : Vol 1. 2013 : 172 – 179.

[15] PODELSKI A, SCHÄF M, WIES T. Classifying bugs with interpolants[C] // International Conference on Tests and Proofs. 2016 : 151 – 168.

[16] LE W, SOFFA M L. Path-based fault correlations[C] // Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering. 2010 : 307 – 316.

[17] MUSKE T, TALLURI R, SEREBRENIK A. Repositioning of static analysis alarms[C] // Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis. 2018 : 187 – 197.

[18] JUNG Y, KIM J, SHIN J, et al. Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis[C] // International Static Analysis Symposium. 2005 : 203 – 217.

[19] KIM S, ERNST M D. Prioritizing warning categories by analyzing software history[C] // Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007). 2007 : 27 – 27.

[20] WILLIAMS C C, HOLLINGSWORTH J K. Automatic mining of source code repositories to improve bug finding techniques[J]. IEEE Transactions on Software Engineering, 2005, 31(6) : 466 – 480.

[21] KREMENEK T, ASHCRAFT K, YANG J, et al. Correlation exploitation in error ranking[J]. ACM SIGSOFT Software Engineering Notes, 2004, 29(6) : 83 – 93.

[22] AYEWAH N, PUGH W, MORGENTHALER J D, et al. Evaluating static analysis defect warnings on production software[C] // Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. 2007 : 1 – 8.

[23] RUTHRUFF J, PENIX J, MORGENTHALER J, et al. Predicting accurate and actionable static analysis warnings[C] // 2008 ACM/IEEE 30th International Conference on Software Engineering. 2008 : 341 – 350.

[24] REYNOLDS Z P, JAYANTH A B, KOC U, et al. Identifying and documenting false positive patterns generated by static code analysis tools[C] // 2017 IEEE/ACM 4th International Workshop on Software Engineering Research and Industrial Practice (SER&IP). 2017 : 55 – 61.

[25] BRUN Y, ERNST M D. Finding latent code errors via machine learning over program executions[C] // Proceedings. 26th International Conference on Software Engineering. 2004 : 480 – 490.

[26] HECKMAN S, WILLIAMS L. A model building process for identifying actionable static analysis alerts[C] // 2009 International conference on software testing verification and validation. 2009 : 161 – 170.

[27] LIANG G, WU L, WU Q, et al. Automatic construction of an effective training set for prioritizing static analysis warnings[C] // Proceedings of the IEEE/ACM international conference on Automated software engineering. 2010 : 93 – 102.

[28] HANAM Q, TAN L, HOLMES R, et al. Finding patterns in static analysis alerts: improving actionable alert ranking[C] // Proceedings of the 11th working conference on mining software repositories. 2014 : 152 – 161.

[29] FLYNN L, SNAVELY W, SVOBODA D, et al. Prioritizing alerts from multiple static analysis tools, using classification models[C] // 2018 IEEE/ACM 1st International Workshop on Software Qualities and their Dependencies (SQUADE). 2018 : 13 – 20.

[30] ZHANG Y, XING Y, GONG Y, et al. A variable-level automated defect identification model based on machine learning[J]. Soft Computing, 2020, 24(2) : 1045 – 1061.

[31] GIFFHORN D, HAMMER C. Precise analysis of java programs using joana[C] // 2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation. 2008 : 267 – 268.

[32] LI Z, ZOU D, XU S, et al. Vuldeepecker: A deep learning-based system for vulnerability detection[J]. arXiv preprint arXiv:1801.01681, 2018.

[33] SESTILI C D, SNAVELY W S, VANHOUDNOS N M. Towards security defect prediction with AI[J]. arXiv preprint arXiv:1808.09897, 2018.

[34] ALLAMANIS M, BARR E T, DEVANBU P, et al. A survey of machine learning for big code and naturalness[J]. ACM Computing Surveys (CSUR), 2018, 51(4) : 1 – 37.

[35] KAMIYA T, KUSUMOTO S, INOUE K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code[J]. IEEE transactions on software engineering, 2002, 28(7) : 654 – 670.

[36] SAJNANI H, SAINI V, SVAJLENKO J, et al. Sourcerercc: Scaling code clone detection to big-code[C] // Proceedings of the 38th International Conference on Software Engineering. 2016 : 1157 – 1168.

[37] JIANG L, MISHERGHI G, SU Z, et al. Deckard: Scalable and accurate tree-based detection of code clones[C] // 29th International Conference on Software Engineering (ICSE'07). 2007 : 96 – 105.

[38] RAYCHEV V, VECHEV M, YAHAV E. Code completion with statistical language models[C] // Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2014 : 419 – 428.

[39] PRADEL M, SEN K. Deepbugs: A learning approach to name-based bug detection[J]. Proceedings of the ACM on Programming Languages, 2018, 2(OOPSLA) : 1 – 25.

[40] WHITE M, TUFANO M, VENDOME C, et al. Deep learning code fragments for code clone detection[C] // 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). 2016 : 87 – 98.

[41] MOU L, LI G, ZHANG L, et al. Convolutional neural networks over tree structures for programming language processing[C] // Thirtieth AAAI conference on artificial intelligence. 2016.

[42] WEI H, LI M. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code.[C] // IJCAI. 2017 : 3034 – 3040.

[43] ALLAMANIS M, BROCKSCHMIDT M, KHADEMI M. Learning to represent programs with graphs[J]. arXiv preprint arXiv:1711.00740, 2017.

[44] ZHAO G, HUANG J. Deepsim: deep learning code functional similarity[C] // Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2018 : 141 – 151.

[45] TUFANO M, WATSON C, BAVOTA G, et al. Deep learning similarities from different representations of source code[C] // 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR). 2018 : 542 – 553.

[46] TRIPP O, GUARNIERI S, PISTOIA M, et al. Aletheia: Improving the usability of static security analysis[C] // Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. 2014 : 762 – 774.

[47] GOLDBERG Y. Neural network methods for natural language processing[J]. Synthesis lectures on human language technologies, 2017, 10(1) : 1 – 309.

[48] GERS F A, SCHMIDHUBER J, CUMMINS F. Learning to forget: Continual prediction with LSTM[J]. Neural computation, 2000, 12(10): 2451 – 2471.

[49] HOCHREITER S, SCHMIDHUBER J. Long short-term memory[J]. Neural computation, 1997, 9(8): 1735 – 1780.

[50] MANDIC D, CHAMBERS J. Recurrent neural networks for prediction: learning algorithms, architectures and stability[M]. [S.l.]: Wiley, 2001.

[51] GORI M, MONFARDINI G, SCARSELLI F. A new model for learning in graph domains[C] // Proceedings. 2005 IEEE international joint conference on neural networks: Vol 2. 2005: 729 – 734.

[52] SCARSELLI F, GORI M, TSOI A C, et al. The graph neural network model[J]. IEEE transactions on neural networks, 2008, 20(1): 61 – 80.

[53] WANG J, WANG S, WANG Q. Is there a ”golden” feature set for static warning identification?: an experimental evaluation[C] // the 12th ACM/IEEE International Symposium. 2018.

[54] WEISER M. Program slicing[J]. IEEE Transactions on software engineering, 1984(4): 352 – 357.

[55] PUTRA I S, RUKMONO S A, PERDANA R S. Abstract Syntax Tree (AST) and Control Flow Graph (CFG) Construction of Notasi Algoritmik[C] // 2021 International Conference on Data and Software Engineering (ICoDSE). 2021: 1 – 6.

[56] COOPER K D, HARVEY T J, WATERMAN T. Building a Control-flow Graph from Scheduled Assembly Code[J], 2003.

[57] CUNNINGHAM P, DELANY S J. k-Nearest neighbour classifiers-A Tutorial[J]. ACM Computing Surveys (CSUR), 2021, 54(6): 1 – 25.

[58] LEUNG K M. Naive bayesian classifier[J]. Polytechnic University Department of Computer Science/Finance and Risk Engineering, 2007, 2007: 123 – 156.

[59] LIAW A, WIENER M, OTHERS. Classification and regression by randomForest[J]. R news, 2002, 2(3): 18 – 22.

[60] QUINLAN J R. C4. 5: programs for machine learning[M]. [S.l.]: Elsevier, 2014.

[61] ANVIK J. Assisting bug report triage through recommendation[D]. [S.l.]: University of British Columbia, 2007.

[62] STOCKTON D J, KHALIL R A, MUKHONGO L M. Cost model development using virtual manufacturing and data mining: part II—comparison of data mining algorithms[J]. The International Journal of Advanced Manufacturing Technology, 2013, 66(9): 1389 – 1396.

[63] COLLOBERT R, WESTON J. A unified architecture for natural language processing: Deep neural networks with multitask learning[C] // Proceedings of the 25th international conference on Machine learning. 2008: 160 – 167.

[64] MIKOLOV T, SUTSKEVER I, CHEN K, et al. Distributed representations of words and phrases and their compositionality[J]. Advances in neural information processing systems, 2013, 26.

[65] KINGMA D P, BA J. Adam: A method for stochastic optimization[J]. arXiv preprint arXiv:1412.6980, 2014.

简历与科研成果

基本信息

葛宇，男，汉族，1998 年 4 月出生，江苏省扬州人。

教育背景

2016.9 — 2020.6 南京大学软件学院 硕士

2020.9 — 2022.6 南京大学软件学院 本科

攻读工程硕士学位期间完成的学术成果

1. Ge X, Fang C, Qian M, **Ge Y** et al. Locality-based security bug report identification via active learning[J]. Information and Software Technology, 2022: 106899.
2. 房春荣、葛修婷、**葛宇**、陈振宇，“一种基于排序学习的代码漏洞误报检测方法”，申请号：202111471775.3，已受理。
3. 房春荣、刘子夕、葛修婷、**葛宇**、李宁、段嘉琪，“一种基于深度学习的代码漏洞误报自动化识别与过滤方法”，申请号：2020104872279，已受理。
4. 房春荣、**葛宇**、刘子夕、葛修婷、钱美缘、李宁，“一种代码漏洞扫描与定位的方法”，申请号：2020104872048，已受理。
5. 房春荣、葛修婷、刘子夕、**葛宇**、李林昱，一种基于人机协同的代码漏洞智能检测方法，申请号：2020104872033，已受理。
6. 房春荣、钱美缘、葛修婷、刘子夕、**葛宇**、段嘉琪、李宁一种代码漏洞智能检测的数据收集方法，申请号：2020104871632，已受理。