软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.000000]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于深度学习的 API 误用缺陷检测*

汪 昕, 陈 驰, 赵逸凡, 彭 鑫, 赵文耘

(复旦大学 软件学院,上海 201203)

(复旦大学 上海市数据科学重点实验室,上海 201203)

通讯作者: 彭鑫, E-mail: pengxin@fudan.edu.cn

摘 要: 开发人员经常需要使用各种应用程序编程接口（Application Programming Interface,简称 API）来复用已有的软件框架、类库等.由于 API 自身的复杂性、文档资料的缺失等原因,开发人员经常会误用 API.从而导致代码缺陷.为了自动检测 API 误用缺陷,需要获得 API 使用规约并根据规约对 API 使用代码进行检测.然而,可用于自动检测的 API 规约难以获得,而人工编写并维护的代价又很高.针对以上问题,本文将深度学习中的循环神经网络模型应用于 API 使用规约的学习及 API 误用缺陷的检测.本文在大量的开源 Java 代码基础上,通过静态分析构造 API 使用规约训练样本,同时利用这些训练样本搭建循环神经网络学习 API 使用规约.在此基础上,本文针对 API 使用代码进行基于上下文的语句预测,并通过预测结果与实际代码的比较发现潜在的 API 误用缺陷.本文对所提出的方法进行实现并针对 Java 加密相关的 API 及其使用代码进行了实验评估,结果表明该方法能够在一定程度上实现 API 误用缺陷的自动发现.

关键词: API 误用;使用规约;缺陷检测;深度学习
中图法分类号: TP311

中文引用格式: 汪昕,陈驰,赵逸凡,彭鑫,赵文耘.基于深度学习的 API 误用缺陷检测.软件学报. http://www.jos.org.cn/1000-9825/ 0000.htm

英文引用格式: Wang X, Chen C, Zhao YF, Peng X, Zhao WY. Deep Learning based API-Misuse bug detection. Ruan Jian Xue Bao/Journal of Software, 2018 (in Chinese). http://www.jos.org.cn/1000-9825/0000.htm

Deep Learning Based API-Misuse Bug Detection

WANG Xin, CHEN Chi, ZHAO Yi-Fan, PENG Xin, ZHAO Wen-Yun

(School of Software, Fudan University, Shanghai 201203, China)

(Shanghai Key Laboratory of Data Science, Fudan University, Shanghai 201203, China)

Abstract: Developers often need to use various Application Programming interfaces (API) to reuse existing software frameworks, class libraries, and so on. Because of the complexity of the API itself, or the lack of documentation, developers often make some API misuses, which can lead to some code defects. In order to automatically detect API misuse defects, the API use specification is required and the API is tested according to the specification. However, API specifications that can be used for automatic detection are difficult to obtain, and the cost of manual writing and maintenance is high. To address the issue, this paper applies the Recurrent Neural Network model of Deep Learning to the task of learning API use specifications and the task of detecting the API misuse defect. In this paper, based on a large number of open source Java code, we extract the training sample of API use specification based of static analysis method, and then use the training sample to set up the recurrent neural network to learning API use specification. On this basis, this paper makes a context-based prediction on the API use code, and finds out the potential API misuse defects by comparing the prediction results with the actual code. In this paper,

* 基金项目: 国家重点研发计划(2016YFB1000801);
Foundation item: The National Key Research and Development Program of China (2016YFB1000801);
收稿时间: 0000-00-00; 修改时间: 0000-00-00; 采用时间: 0000-00-00; jos 在线出版时间: 0000-00-00
CNKI 在线出版时间: 0000-00-00

we implement the method above, and evaluate this method with experiments about Java encryption related APIs and their used code. The results show that to a certain extent our approach has the ability to automatically detect API misuse defects.

Key words: API Misuse; Usage Specification; Bug Detection; Deep Learning

1 引言

在软件开发过程中开,发人员经常需要使用各种应用程序编程接口(Application Programming Interface,简称API)来复用已有的软件框架、类库,以节省软件开发时间,提高软件开发效率.但由于 API 自身的复杂性、文档资料的缺失或自身的疏漏等原因[1,2,3,4]开发人员经常会误用 API.API 的误用情形多种多样[5],例如多余的 API 调用、遗漏的 API 调用、错误的 API 调用参数、缺少前置条件判断、忽略异常处理等.这些 API 误用在实际项目中常常导致了功能性错误、性能问题、安全漏洞等代码缺陷[5,6].例如,在使用文件流 API 时,如果最后遗漏了对文件流进行关闭的 API 调用,那么将导致内存泄露问题.

为了自动检测 API 误用缺陷,需要获得 API 使用规约[7]并根据规约对 API 使用代码进行检测.然而,可用于自动检测的 API 规约难以获得,而人工编写并维护的代价又很高. 相关研究工作[8,9,10]关注于利用数据挖掘、统计语言模型等方法自动挖掘或学习 API 使用规约并用于缺陷检测,但存在合成能力不足等问题.

本文将深度学习中的循环神经网络模型应用于 API 使用规约的学习及 API 误用缺陷的检测.在大量的开源 Java 代码基础上,通过静态分析构造 API 使用规约训练样本,同时利用这些训练样本搭建循环神经网络结构学习 API 使用规约.在此基础上,本文针对 API 使用代码进行基于上下文的语句预测,并通过预测结果与实际代码的比较发现潜在的 API 误用缺陷.本文对所提出的方法进行实现并针对 Java 加密相关的 API 及其使用代码进行了实验评估,结果表明该方法能够在一定程度上实现 API 误用缺陷的自动发现.

本文共分为五节.第一节对本文使用的深度学习背景知识进行说明;第二节介绍相关工作;第三节对本文所提出的方法及其实现进行说明;第四节介绍本文中设计的两个实验——深度学习模型训练实验和 API 误用缺陷检测实验的设计与结果分析;最后在第五节对本文进行总结与展望.

2 背景知识

深度学习（Deep Learning,简称 DL）是一类通过连续多层变换的非线性处理单元,对数据进行复杂特征提取并通过这些组合特征解决问题的机器学习算法[11].

2.1 循环神经网络

循环神经网络（Recurrent Neural Networks,简称 RNN）,属于深度学习技术的一个重要分支.通过对数据中的时序信息的利用以及对数据中语义信息的深度表达,RNN 在处理和预测序列数据上实现了突破,并在语音识别、语言模型、机器翻译等方面发挥出色[12,13,14,15].

2.2 长短时记忆网络

RNN 利用状态值保存迭代计算时的历史信息, 利用时序信息,辅助当前决策.但简单的 RNN 存在长期依赖（long-term dependencies）问题. 1997 年,Hochreiter 和 Schmidhuber 提出的长短时记忆网络（Long Short Term Memory, 简称 LSTM）[16,17] ,经过计算[16,17],能够进行前向传播,并对历史信息进行遗忘,对输入信息进行状态更新,有效解决了长期依赖问题.

2.3 深层循环神经网络

深层循环神经网络（Deep RNN）是 RNN 的一种变种,通过拓展浅层循环体,设置多个循环层,将每层 RNN 的输出作为下一层 RNN 的输入,进一步处理抽象,增强 RNN 从输入中提取高维度抽象信息的能力[12].

2.4 神经网络模型的优化

2.4.1 损失函数

损失函数定义了深度学习神经网络模型的效果和优化目标.本文将神经网络模型应用于分类问题,用 N 个神经元的输出表示 N 个不同的 API 调用的评判概率.应用于分类问题的常用损失函数是交叉熵(cross entropy),它表示了两个概率分布之间的距离,可用于计算正确答案概率分布与预测答案概率分布之间的距离.在训练过程中,使用 Softmax 回归处理层将神经网络的前向输出转换为大小为 N 的概率分布形式.

2.4.2 优化算法

在神经网络模型中,优化过程分为两个阶段:一是前向传播阶段,在这个阶段,神经网络模型前向计算预测值,并通过预测值与目标值计算损失函数,二是反向传播阶段,在这个阶段, 采用梯度下降法(gradient descent)[18]的思想,通过损失函数,对每个可优化的参数进行梯度计算,同时利用学习率进行参数更新,从而达到优化模型的目的.

在神经网络模型优化过程中,选取适当的初始值、学习率、迭代次数,能进一步调整模型的预测能力.同时,也要注意过拟合问题对模型预测能力的影响——当模型进入过拟合状态,模型的表达能力将局限在训练数据中,反而不利于模型的泛化预测能力.

3 相关工作

不少研究工作关注于利用数据挖掘、统计语言模型等方法自动挖掘或学习 API 使用规约并用于缺陷检测,但存在合成能力不足等问题.这方面有代表性的研究工作包括以下这些.

Li[8]等人研究了基于频繁项集的 API 使用规约挖掘技术,并研发了一个挖掘和检测工具 PR-Miner.PR-Miner 通过将代码库中的函数映射为数据子项,并对代码中频繁共现的数据子项进行统计和挖掘,根据数据子项的频繁共现关系推断 API 调用之间的关联关系.最终,PR-Miner 可以得到以 API 调用的关联关系为基础的使用规约.基于该工具,他们在 Linux kernel、PostgreSQL 等大型软件系统的代码中发现了 23 个确认的 Bug.

Zhong[9]等人研究了基于自然语言处理的 API 使用规约提取技术,并研发了一个 API 使用规约提取工具 Doc2Spec.Doc2Spec 通过分析自然语言编写的 API 文档,提取相关的 API 使用规约.基于该工具,他们分析了 J2SE4、J2EE5、JBoss6、iText7 以及 Oracle JDBC driver8 这 5 个 Java 库的 Javadocs,提取出 API 使用规约,并基于这些 API 使用规约在真实代码中人工发现了 35 个确认的 Bug.

Wang[10]等人研究了 N-gram 语言模型,并研发了一个缺陷检测工具 Bugram.Bugram 利用 N-gram 语言模型,基于 API 调用 token 只与在它之前的 n 个 token 相关的假设,对软件项目中出现的 API 调用 token 序列进行出现概率的计算,同时将较低概率的 token 序列的出现与程序中 API 的异常调用、误用和特殊用法联系起来,以此进行缺陷的自动化检测.基于该工具,他们在 16 个开源 Java 项目中发现了 25 个确认的 Bug 以及 17 个可能存在的代码重构建议.

然而,这些研究中提出的技术方法存在一定的局限性.频繁项集挖掘技术关注于挖掘代码中数据子项的频繁共现关系,但对 API 使用的顺序信息没有加以利用;而 N-gram 语言模型利用到了 API 使用的顺序信息,但是对多样和变长的代码上下文存在模型的合成能力不足的问题.

本文将循环神经网络模型应用于 API 使用规约的学习及 API 误用缺陷的检测.循环神经网络模型能有效提取序列数据中的抽象特征,以概率分布的形式进行预测,能有效处理多样和变长的上下文,具有较强的合成能力.改进后的循环神经网络有效地解决了自然语言处理中的长期依赖问题,被广泛应用于模式识别等领域.因此我们期望该模型也能有效学习 API 调用组合、顺序及控制结构等方面的使用规约.

4 方法与实现

4.1 方法概览

本文所提出的 API 误用缺陷检测方法概览如图 1 所示,其中主要包括三个阶段：训练数据构造、模型训练与预测、缺陷检测.

在训练数据构造阶段,基于大量开源 Java 代码数据集,对源代码进行静态分析,构造抽象语法树,并进一步构造为 API 语法图,用于 API 调用序列的抽取,最后,将 API 调用序列构造为深度学习模型需要的训练数据;在模型训练与预测阶段,训练深度学习模型,基于该模型和 API 使用的上下文对 API 调用序列中某处的 API 调用进行预测,得出该位置的 API 调用概率列表;在缺陷检测阶段,从待检测代码中抽取用于预测的 API 调用序列,基于模型训练与预测阶段的模型预测对每个位置进行 API 调用预测,并获取每个位置的 API 调用概率列表,将原代码中的 API 调用序列与预测得出的概率列表进行比对,得出最终的 API 误用检测报告.

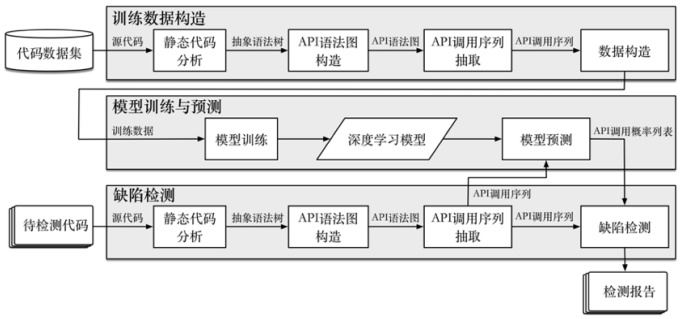


Fig.1 Overview of our approach

图 1 方法概览

4.2 训练数据构造

在训练数据构造阶段,将数据从源代码的形式,依次构造为抽象语法树、API 语法图,并基于构造得出的 API 语法图,提取出与 API 调用相对应的 API 调用序列,随后,将 API 调用序列构造为可用于模型训练的训练数据.如图 1 所示,训练数据构造阶段分为四个子阶段：静态代码分析,API 语法图构造,API 调用序列抽取,数据构造.

4.2.1 静态代码分析

抽象语法树（Abstract Syntax Tree,简称 AST）将 Java 源代码映射成树状结构,是静态解析源代码的有效工具.在静态代码分析阶段,本文使用 JavaParser[19]对源代码文件进行解析,并获取表示源代码语法结构的 AST.

4.2.2 API 语法图构造

在 API 语法图构造阶段,在 AST 的基础上进行进一步解析构造,经过静态代码分析和 API 语法图构造,代码从源代码形式转化为 API 语法图形式.如图 2 所示,展示了本文中一个 API 语法图抽取的例子.

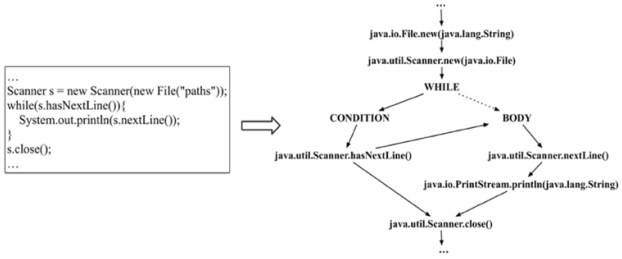


Fig.2 Example for API Graph Extraction

图 2 API 语法图抽取例子

图中的代码通过 Scanner API 读取每行文件内容并打印到输出控制台中,最后关闭 Scanner,释放资源.在该代码中使用到了 java.io.File,java.util.Scanner,java.io.PrintStream 这些 API,经过转化,抽取出的 API 语法图如图中图状结构所示.

在构造 API 语法图的过程中涉及到如下定义:

(1) 节点

节点分为三种类型:

- 方法节点,表示 API 语法图的来源(文件路径、类、方法);
- API 节点,表示 API 对象创建、API 方法调用、API 类变量访问;
- 控制节点,表示控制结构(如 IF,ELSE,WHILE,FOR,FOREACH,TRY 等).

(2) 边

节点之间可以由有向边连接,方向从父节点指向子节点.边分为两种类型:

- 顺序边,表示流(控制流)的方向;
- 控制边,表示控制结构之间的顺序关系(如 IF -> ELSE).

(3) API 语法图

一个根节点,能构成一个最简的 API 语法图.

边从父节点指向子节点,没有子节点的节点称为叶子节点.

向 API 语法图中的一个节点向该图中另一个节点加一条边,图中不构成环路,这个图仍是一个 API 语法图.

向 API 语法图中加入一个节点,由图中任意一个节点向该节点加一条边,图中不构成环路,这个图仍是一个 API 语法图.

两个 API 语法图,从一个 API 语法图的叶子节点出发,向另一 API 语法图的根节点加边,称为移植.移植后,得到组合了两个 API 语法图的 API 语法图.

根据定义进行 UML 建模,设计 API 语法图类以及相关类如图 3 所示.

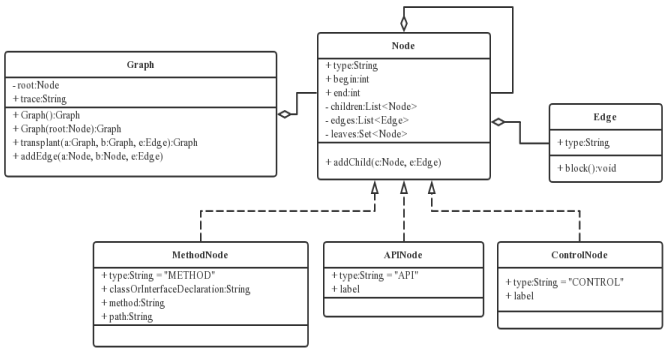


Fig.3 Class Diagram for API Graph Extraction

图 3 API 语法图相关类图

其中:

• Node 表示 API 语法图上的节点,分为三种节点类型,包括方法节点(MethodNode)、API 节点(APINode)以及控制节点(ControlNode).

• Edge 表示父节点连接子节点的边,可以为顺序类型边或者控制结构类型边.假设从图上从一个节点出发,深度遍历找到所有没有子节点与之连接的节点,这些节点为这个节点的叶子节点(leaves).

• Graph 表示 API 语法图,该图上的节点由 Node 表示,其中,一个特殊的根节点 root 表示一个 API 语法图的开始.从 Graph a 的叶子节点到 Graph b 的根节点都加上边,这个操作称为移植(transplant).

在 API 语法图构造阶段,基于 AST 以及控制流分析,将源代码抽象表示为 API 语法图,流程的具体步骤如下:

(1) 基于 JavaParser,静态分析代码为 AST;

(2) 基于 JavaParser 对每个类中的方法进行抽取,将信息存储到方法节点中,并以该节点为根节点,构建 API 语法图 g;

(3) 对每个方法根据以下规则,迭代地构建 API 语法图 g':

a) 如果当前处理语句中有嵌套的语句,先构造出嵌套在内层的语句的 API 语法图,移植到 g'上,再迭代地根据后续规则构造出外层的语句的 API 语法图,移植到 g'上;

b) 如果当前处理语句是 API 对象创建(API Object Creation),则构造 API 节点,并以"Full Class Name .new([Full Parameter Class Types])"的形式表示节点,并将以该节点为根节点的 API 语法图移植到 g'上;

c) 如果当前处理语句是 API 方法调用(API Method Call),则构造 API 节点,并以"Full Method Name([Full Parameter Class Types])"的形式表示节点,并将以该节点为根节点的 API 语法图移植到 g'上;

d) 如果当前处理语句是 API 类变量访问(API Field Access),则构造 API 节点,并以"Full Class Name .Field"的形式表示节点,并将以该节点为根节点的 API 语法图移植到 g'上;

e) 如果当前处理语句是控制结构,则根据以下主要规则,构建相应的 API 语法图:

i. "if"(判断结构),构造控制节点"IF",并设置为 g'的根节点.根据当前处理语句的子结构,分别构造以控制节点"CONDITION"、"THEN"、"ELSE" 为根节点的三个 API 语法图,并用控制边将这三个 API 语法图移植到 g'上.

ii. "while"(循环结构),构造控制节点"WHILE",并设置为 g'的根节点.根据当前处理语句的子结构,分别构造以控制节点"CONDITION"、"BODY"为根节点的两个 API 语法图,并用控制边将这两个 API 语法图移植到 g'上.

iii. "try"(异常处理结构),构造控制节点"TRY",并设置为 g'的根节点.根据当前处理语句的子结构,分别构造以控制节点"TRYBLOCK"、"CATCH"、"FINALLY"为根节点的 API 语法图,并用控制边将这些 API 语法图移植到 g'上.

iv. "for"(包含变量初始化、条件判断、变量更新的循环结构),构造控制节点"FOR",并设置为 g'的根节点.根据当前处理语句的子结构,分别构造以控制节点"INITIALIZATION"、"COMPARE"、"BODY"、"UPDATE"为根节点的 API 语法图,并用控制边将这些 API 语法图移植到 g'上.

v. "for each"(循环集合访问结构),构造控制节点"FOREACH",并设置为 g'的根节点.根据当前处理语句的子结构,分别构造以控制节点"VARIABLE"、"ITERABLE"、"BODY"为根节点的 API 语法图,并用控制边将这些 API 语法图移植到 g'上.

(4) 将 API 语法图 g'移植到 API 语法图 g 中,获得每个类中方法对应的 API 语法图.最终获取的 API 语法图将作为下一阶段的基础.

4.2.3 API 调用序列抽取

定义 API 调用序列(API Sequence)如下:

给定 API 语法图,从根节点出发,根据顺序边以及表示顺序的控制边对 API 语法图进行深度遍历得到的节点标签序列,称为这个 API 语法图上的 API 调用序列.表示顺序的控制边有以下几种:IF->CONDITION,WHILE->CONDITIO,TRY->TRYBLOCK,FOR->INITAILIZATION,FOREACH->VARIABLE.

例如,在图 2 的例子中,经过 API 语法图构造以及 API 调用序列抽取,得到以下两个 API 调用序列:

- "java.io.File.new(java.lang.String)->java.util.Scanner.new(java.io.File)->WHILE->CONDITION->java.util.Scanner.hasNextLine()->BODY->java.util.Scanner.nextLine()->java.io.PrintStream.println(java.lang.String)->jav

a.util.Scanner.close()->EOS”

- “java.io.File.new(java.lang.String)->java.util.Scanner.new(java.io.File)->WHILE->CONDITION->java.util.Scanner.hasNextLine()->java.util.Scanner.close()->EOS”

在 API 调用序列抽取阶段,基于 API 语法图,抽取 API 语法图中的 API 调用序列.主要思想是对 API 语法图进行深度遍历,提取所有存在 API 调用节点的 API 调用序列作为与该 API 语法图对应的 API 调用序列集合.所有提取的 API 调用序列以 EOS 控制节点结束.

4.2.4 训练数据产生

训练数据构造阶段将 API 调用序列转换为训练数据.该阶段包括了词汇表的建立以及训练数据的构造两个部分.

(1) 构建词汇表.为了将 API 调用序列转化为深度学习模型能够统一识别的序列输入,将原始数据集中的代码片段转化为 API 调用序列后,经过统计序列中出现的 API 调用词频,建立 API 调用与 API 调用编号一一对应的词汇表,并持久化到本地的词汇表文件中.

(2) 构造训练数据.如图 4 中构造训练数据算法流程图所示,基于词汇表将 API 调用序列转换为 API 调用编号序列,其中每个 API 调用的编号与词汇表中该调用的行号对应.对构造好的编号序列进行遍历,构造<API 调用前文编号序列,API 调用编号>形式数据,并作为训练数据持久化至本地训练数据文件中,方便之后的模型训练和预测.这里的 API 调用前文指的是在该序列中 API 调用之前的所有 API 调用组成的序列,API 调用前文的编号序列形式称为 API 调用前文编号序列.

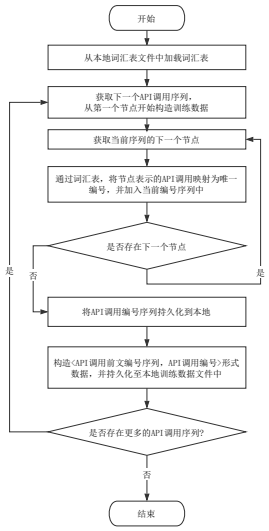


Fig.4 Flowchart of Training Data Build

图 4 构造训练数据算法流程图

4.3 模型训练与预测

在模型训练与预测阶段,基于 TensorFlow 框架,使用 Python3 语言搭建深度学习模型对训练数据构造中获得的训练数据进行学习,得到训练好的深度学习模型,并利用该模型对特定时间步上的 API 调用进行预测,得到该位置上的 API 调用概率列表.该阶段主要分为模型训练和模型预测两个阶段.

4.3.1 模型训练

本文中采用的深度学习模型为深层循环网络结构配合长短时记忆网络的应用,这里简称为深层长短时记忆循环网络（Deep LSTM）.结构示意如图 5 所示.示意图中采用了双层深层循环神经网络的结构,在每个循环

体结构中应用 LSTM 模型,将每个时间步上的 API 调用序列的标签作为一个词,Deep LSTM 的输入为包含t个时间步的词向量,表示前t个 API 调用序列的输入,输出取第t个时间步的输出,表示基于前文内容,下一个 API 调用序列上的节点可能情况的概率分布.

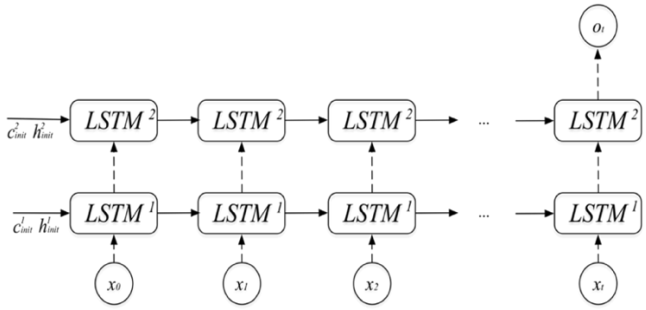


Fig.5 Structure of Deep LSTM

图 5 Deep LSTM 结构示意图

模型的搭建选取深度学习框架 TensorFlow 1.6.0 作为实现框架.

模型的计算流程如下:

(1) 经过词向量层(word embedding),将输入的 API 调用前文编号序列中的每个 API 调用编号嵌入到一个实数向量中,以此降低输入的维度,同时增加语义信息.

(2) 经过词向量层转化的输入向量经过 dropout 层,丢弃部分信息以增强模型的健壮性.

(3) 使用 TensorFlow 提供的动态 RNN 接口实现神经网络主要结构,其中包含了深层 LSTM 处理单元.

(4) 循环神经网络的输出需要经过一个全连接层(dense)转化,与词表大小保持一致,这个与词表大小一致的输出称为 logits,随后,logits 经过 Softmax 处理以及交叉熵计算,可以分别得到 API 调用的概率分布以及损失值,TensorFlow 将基于给定的优化函数,对参数进行优化.

(5) 同时,在训练过程中加入了预测正确率(accuracy)作为训练时的反馈,同时也作为优化训练模型的参考.

至此,我们搭建并训练了一个应用于 API 调用序列预测的深度学习模型.

4.3.2 模型预测

在模型预测阶段,基于模型训练中的训练好的深度学习模型,并将用于预测的 API 调用前文序列作为输入,预测下一位置上的 API 调用的概率分布,将这个概率分布进行排序,得出下一位置上的 API 调用概率列表.该 API 调用概率列表将作为之后缺陷检测的重要部分.

4.4 缺陷检测

本阶段的基础是训练数据构造阶段和模型训练与预测阶段.

如图 1 所示,缺陷检测阶段经过训练数据构造阶段的静态代码分析、API 语法图构造、API 调用序列抽取,得到原代码中的 API 调用序列,基于模型训练与预测阶段的模型预测,对 API 调用序列中除了具有固定语法顺序搭配的控制流结构(例如 IF->CONDITION 中 IF 后紧跟的 CONDITION 所在位置,WHILE->CONDITION 中 WHILE 后紧跟的 CONDITION 所在位置等等)之外的每个位置进行 API 调用预测并获取每个位置的 API 调用概率列表,将原代码中每个 API 调用序列上的每个位置的 API 调用与预测得出的 API 调用概率列表进行比对,应用定义好的代码缺陷检测规则,报告出候选的代码缺陷,得出最终的 API 误用检测报告.

缺陷检测实现部分涉及不同的实现语言——Java 和 Python,为了模型之间较方便的协作调用,在本文的实现中采用如下技术框架进行模块之间的沟通:基于 Python 语言中的 Web 端框架 Flask[20]搭建 Web 服务端,将 API 调用概率列表的预测包装为 Web API 供客户端调用.

本文中实现的缺陷检测算法流程如图 6 所示,算法中的具体步骤如下:

(1) 从待检测的源代码中抽取 API 语法图;

(2) 从 API 语法图中抽取 API 调用序列 ;

(3) 对每一个 API 调用序列,从第二个位置开始,进行缺陷检测;

(4) 基于检测位置的 API 调用前文,预测该位置可能出现的 API 调用概率列表;

(5) 判断该位置的 API 调用在 API 调用概率列表中的排名是否在可接受范围（该范围在算法开始前由人工定义）内,如果该 API 调用的出现在可接受范围内,则继续检测下一位置,如果该 API 调用的出现超出可接受范围,则作为一个缺陷并进行该位置的缺陷报告,缺陷报告包括了代码缺陷在源代码中的位置（起止行数）以及对应的 API 调用;

(6) 结束检测后,可根据缺陷报告进行进一步的评估.

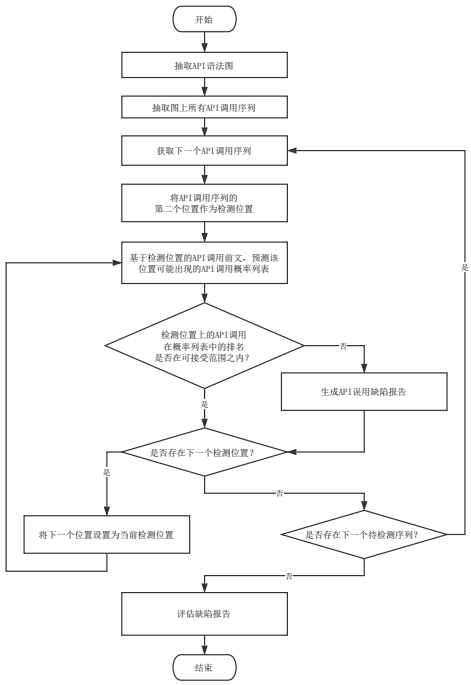


Fig.6 Flowchart of Bug Detection

图 6 缺陷检测算法流程图

在实验过程中,将重点对算法中 API 调用在 API 调用概率列表中可接受范围的设定进行实验.

5 实验评估

本文中的实验评估分为两部分.首先是深度学习模型训练,目的在于尝试优化深度学习模型对 API 调用序列预测的准确性和可靠性.其次是代码缺陷检测实验,目的在于评估检测深度学习模型在 API 误用缺陷检测上的效果和可用性.

5.1 实验对象

JCE（Java Cryptography Extension）是 JDK 提供的一组包,它们提供用于加密、密钥生成和协商以及消息验证码（MAC）算法的框架和实现,JCE 提供的 Java 密码学 APIs（Java Cryptography APIs）在 javax.crypto 包下.研究[21,22,23]说明,Java Cryptography APIs 通过分离开发人员和 API 底层实现细节,开发人员可以轻松地使用加解密技术.这些 API 提供了多种模式和配置选项,但因此,对开发人员来说使用和组合这些 API 组件的任务可

能具有挑战性.

本文中将实验对象聚焦于 Java Cryptography APIs,展示深度学习模型在缺陷检测上的效果和能力.

5.2 深度学习模型训练实验

5.2.1 实验数据

Git 是一个自由开源的分布式版本控制系统,它的设计目的是无论小型还是大型的项目,都能通过 Git 快速高效地管理.GitHub 是一个基于 Git 的大型开源代码托管平台以及版本控制系统,GitHub 上的开源代码是挖掘 API 调用规约的一大数据来源[24,25].

在本文的深度学习模型训练实验中,以“javax.crypto”为关键词,从 GitHub 上收集了 14422 个 java 代码文件,这些文件的最后修改时间在 2018 年 1 月 1 日之前(稳定版本),这些文件组成的训练原始文件总大小为 50MB.从训练原始文件中抽取 38602 条 API 调用序列,构造 388973 条训练数据,词表大小为 1564.

5.2.2 实验设计

深度学习模型的搭建选取深度学习框架 TensorFlow 1.6.0 作为实现框架,语言为 Python,开发 IDE 为 Jupyter-notebook 编辑器,搭建模型为深层循环神经网络和长短时记忆网络的结合.

实验探究隐层大小(HIDDEN_SIZE)、深层循环网络层数(NUM_LAYER)、学习率(LR)以及迭代次数(NUM_EPOCH)对模型效果的影响.对模型效果的评判效果以验证集上的分类准确率(Accuracy)为准.分类准确率为模型预测 Top-1 为目标词的准确率.通过调整深度学习模型的部分参数,对模型进行调整训练,训练集为总数据集的 90%,验证集为总数据集的 10%.

5.2.3 实验模型分析

本次实验最终选择的参数配置为 HIDDEN_SIZE=250,NUM_LAYER=2,LR=0.002,NUM_EPOCH=20.如图 7 所示,横轴为训练迭代次数(Epoch),纵轴为损失(Loss).该模型在训练中的损失不断下降趋于收敛.

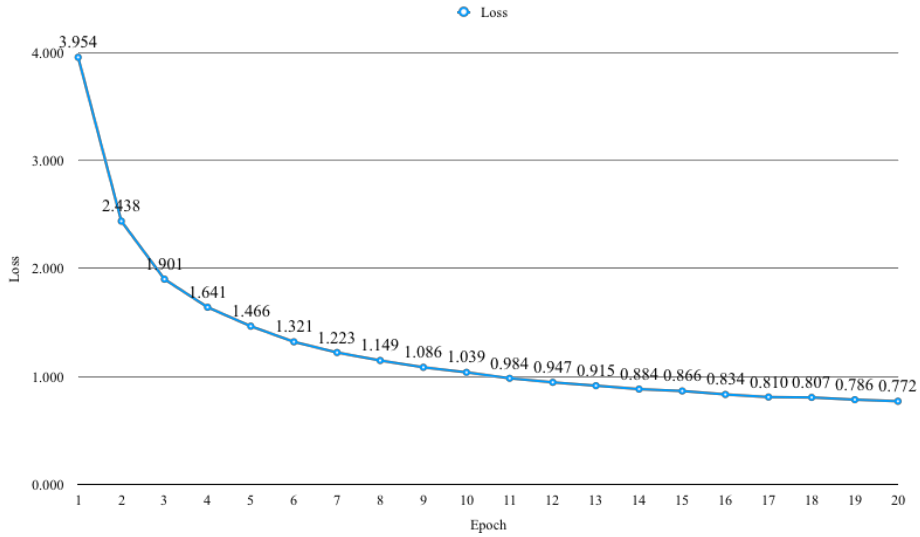


Fig.7 Loss of DL Model

图 7 深度学习模型在训练中的损失趋势

该配置下模型效果如图 8 所示,横轴为迭代次数,纵轴为准确率.

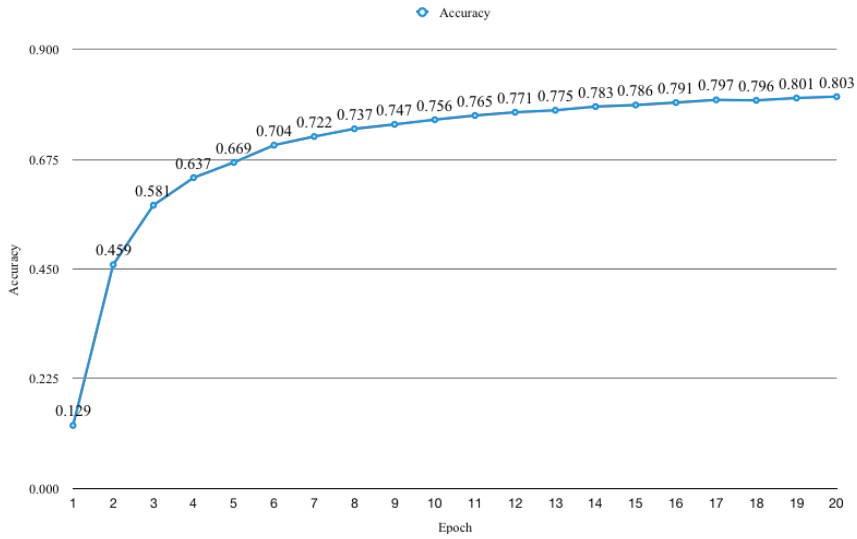


Fig.8 Accuracy of DL Model（HIDDEN_SIZE=250,NUM_LAYER=2,LR=0.002）

图 8 模型准确率（HIDDEN_SIZE=250,NUM_LAYER=2,LR=0.002）

经过 20 次迭代后,模型准确率达到 80.3%,该参数组合在验证集上效果最优,因此最终用于缺陷预测的模型参数为： HIDDEN_SIZE=250,NUM_LAYER=2,LR=0.002,NUM_EPOCH=20.

5.3 代码缺陷检测实验

在该部分进行基于深度学习模型的 API 误用相关代码缺陷检测实验.进行该部分实验前,对实验中的测试标准进行定义：

定义TP为检测文件 API 误用缺陷位置正确的不重复缺陷报告数;

定义FP为检测文件 API 误用缺陷位置错误的不重复缺陷报告数;

定义FN为未进行报告的缺陷报告数;

1. 查准率（Precision）定义为：

$$P = \frac{TP}{TP + FP} \tag{1}$$

2. 召回率（Recall）定义为：

$$R = \frac{TP}{TP + FN} \tag{2}$$

3. 查准率和召回率的调和均值（F1）定义为：

$$F_1 = \frac{2TP}{2TP + FP + FN} \tag{3}$$

查准率和召回率往往体现的是实验结果的一个侧面,比如召回率较高的模型在查准率上不一定很高,因此在本文的实验中,采用二者的调和均值F_1进行 API 误用缺陷检测的实验评价.

5.3.1 实验数据

本次实验中,根据相关文献中 Java Cryptography API 误用的真实数据[5],选用了从 SourceForge 和 GitHub 上的优质项目（大于 10 星）中整理的关于 Java Cryptography APIs 的误用代码片段 8 个,其中包含 API 误用 14 个,将这 14 个 API 误用作为计算模型查准率、召回率以及 F1 值的测试集.实验所用的测试集整理如表 1 所示,表格中标明了每个测试用例的代码来源,API 误用方法来源,误用引入缺陷的具体 API,以及 API 误用的说明,所有测试用例源码可通过代码来源在托管平台上下载.

Table 1 Source information of test case

表 1 测试用例信息

编号	代码来源	API 误用方法	API 误用	API 误用说明
1	https://github.com/alibaba/druid/commit/e10f2849d046265bf17360ab4aa9eb60fd3ab8de	decrypt(PublicKey, String)	javax.crypto.Cipher.init(int,java.security.Key)	An instance of Cipher is used twice (the init() method is called again), which is an invalid operation.
2	https://github.com/alibaba/druid/commit/e10f2849d046265bf17360ab4aa9eb60fd3ab8de	encrypt(byte[], String)	javax.crypto.Cipher.init(int,java.security.PrivateKey)	A call to Cipher.init() may throw an InvalidKeyException.
3	https://github.com/android-rcs/rcsjta/commit/04d84799daa51ed7cc0ad270f0eea51ffaf7a53a#diff-bf160ca00204f2ae4c100aabe57a1dfd	getContributionId(String)	java.lang.String.getBytes()	Exports bytes for Mac.doFinal() without specifying the encoding.
4	https://github.com/android-rcs/rcsjta/commit/04d84799daa51ed7cc0ad270f0eea51ffaf7a53a#diff-bf160ca00204f2ae4c100aabe57a1dfd	getContributionId(String)	javax.crypto.Mac.doFinal(byte[])	Exports bytes for Mac.doFinal() without specifying the encoding.
5	http://sourceforge.net/p/adempiere/svn/1312/tree/trunk/looks/src/org/compiere/util/Secure.java?diff=5139a2ef34309d2ec1827857:1311	encrypt(String)	java.lang.String.getBytes()	A string is converted to bytes without specifying an explicit encoding. The bytes are then passed to Cipher.doFinal().
6	http://sourceforge.net/p/adempiere/svn/1312/tree/trunk/looks/src/org/compiere/util/Secure.java?diff=5139a2ef34309d2ec1827857:1311	encrypt(String)	javax.crypto.Cipher.doFinal(byte[])	A string is converted to bytes without specifying an explicit encoding. The bytes are then passed to Cipher.doFinal().
7	http://sourceforge.net/p/adempiere/svn/1312/tree/trunk/looks/src/org/compiere/util/Secure.java?diff=5139a2ef34309d2ec1827857:1311	decrypt(String)	java.lang.String.new(byte[])	An encrypted message is decrypted and then converted back to a string, without specifying an explicit encoding. The fix specifies the encoding "UTF-8".
8	http://sourceforge.net/p/battleforge/code/878/tree/trunk/de.battleforge/src/java/de/battleforge/util/BFProperties.java?diff=50ee84dee88f3d24b3d975fe:877	setProperty(BFProps, String, boolean)	java.lang.String.new(byte[])	Encoded data is converted into a String for storing, without explicitly specifying an encoding. The fix introduces base64 encoding.
9	http://sourceforge.net/p/battleforge/code/878/tree/trunk/de.battleforge/src/java/de/battleforge/util/BFProperties.java?diff=50ee84dee88f3d24b3d975fe:877	setProperty(BFProps, String, boolean)	java.lang.String.getBytes()	Text is converted to bytes for encoding without an explicit encoding. The bytes are then passed to Cipher.doFinal().
10	http://sourceforge.net/p/battleforge/code/878/tree/trunk/de.battleforge/src/java/de/battleforge/util/BFProperties.java?diff=50ee84dee88f3d24b3d975fe:877	setProperty(BFProps, String, boolean)	javax.crypto.Cipher.doFinal(byte[])	Text is converted to bytes for encoding without an explicit encoding. The bytes are then passed to Cipher.doFinal().
11	http://sourceforge.net/p/battleforge/code/878/tree/trunk/de.battleforge/src/java/de/battleforge/util/BFProperties.java?diff=50ee84dee88f3d24b3d975fe:877	getProperty(BFProps)	java.lang.String.getBytes()	Encoded data is retrieved from a string (from storage) without explicitly specifying an encoding. The bytes are then passed to Cipher.doFinal().
12	http://sourceforge.net/p/battleforge/code/878/tree/trunk/de.battleforge/src/java/de/battleforge/util/BFProperties.java?diff=50ee84dee88f3d24b3d975fe:877	getProperty(BFProps)	javax.crypto.Cipher.doFinal(byte[])	Encoded data is retrieved from a string (from storage) without explicitly specifying an encoding. The bytes are then passed to Cipher.doFinal().
13	http://sourceforge.net/p/battleforge/code/878/tree/trunk/de.battleforge/src/java/de/battleforge/util/BFProperties.java?diff=50ee84dee88f3d24b3d975fe:877	getProperty(BFProps)	java.lang.String.new(byte[])	Decoded data is converted to String without explicitly specifying an encoding.
14	https://sourceforge.net/p/jmrtd/code/51/tree//passporthostapi/src/sos/mrtd/SecureMessagingWrapper.java?diff=5058d727fd48f84fd52d6740:50	readDO8E(DataInputStream, byte[])	EOS	DataOutputStream is left open.

这些测试用例中包含四种类型的 API 误用（不互斥）：

a. 使用了多余的 API 调用;

b. 使用了错误的 API 调用;

c. 遗漏了关键的 API 调用;

d. 忽略了对 API 调用中可能抛出的异常的处理.

其中,测试用例 1 使用了多余的 API 调用;测试用例 2 忽略了对 API 调用中可能抛出的异常的处理;测试用

例 1、3、5、7、8、9、11、13 使用了错误的 API 调用;测试用例 4、6、10、12、14 遗漏了关键的 API 调用.

5.3.2 实验设计

人工整理 API 误用测试用例,将每个 API 误用标注成标准化的格式“文件路径,方法,API 误用缺陷位置,误用 API”,用于计算查准率、召回率和 F1 值.

在进行 API 误用缺陷检测时, DeepLSTM 将预测出 API 调用序列上某个位置上的 API 调用概率列表,为进行缺陷检测,定义 API 调用在 API 调用概率列表中可以接受的排名范围,在本文中称为可接受阈值(Acceptable Threshold).假设实验中的缺陷检测模型的可接受水平设置为 k,那么,当 API 调用在 API 调用概率列表中的排名在 Top-1 到 Top-k 之间时(包括 Top-k),认为该 API 调用在本次 API 误用缺陷检测中处于 API 调用的可接受范围内,反之认为该 API 调用属于 API 误用代码缺陷.

为验证模型的有效性,设置以下模型作为实验的对比模型:

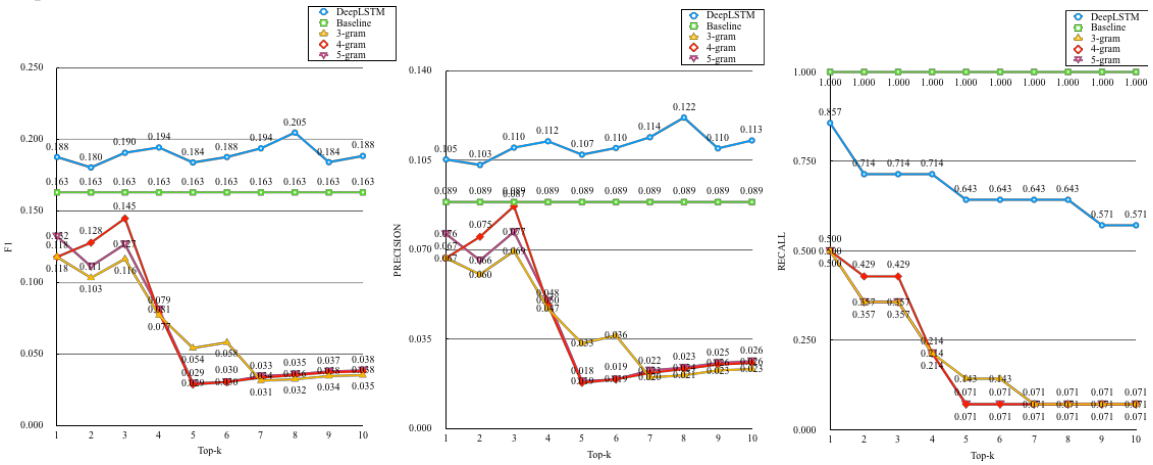
(1) 基准模型(Baseline).该模型假设 API 调用序列上的每个位置都可能是潜在的 API 误用,该假设覆盖到所有的 API 误用缺陷,召回率恒定为 1,在测试集上的查准率为 0.0886, F1 值为 0.163.

(2) N-gram 检测模型.Bugram[10] 基于 N-gram 语言模型实现了对存在 API 误用的异常序列的检测,本文关注于对 API 调用序列上某一位置可能存在的 API 误用的检测.因此本文实现 Bugram 应用的 N-gram 预测模型作为对比实验,使用与深度学习模型训练时相同的 API 调用序列,训练 N-gram 模型,基于前 N 个词元预测下一个词元.实验中取 N={3,4,5}分别对应 3-gram 模型,4-gram 模型和 5-gram 模型,并进行 Jelinek-Mercer 平滑处理[26].N-gram 检测模型应用 N-gram 预测模型对 API 调用序列上某个位置的 API 调用概率列表进行预测,同样的,应用可接受阈值(Top-k)进行缺陷检测.

在本实验中比较不同模型的 API 误用缺陷检测效果,并探究可接受阈值的取值对代码缺陷检测模型的 F1 值的影响.

5.3.3 实验结果分析

经测试,API 误用缺陷检测的实验结果如图 9 所示,图中折线表示不同的实验模型,横轴表示可接受阈值(Top-k)的取值,图 9(a)是各模型的 F1 值,图 9(b)是各模型的查准率,图 9(c)是各模型的召回率.



(a) F1 score (b) Precision (c)Recall

Fig.9 F1 score,Precision and Recall of different API misuse bug detection models

图 9 API 误用缺陷检测实验结果

从结果中可以看出:

在一定范围内,DeepLSTM 的查准率均高于基准模型且召回率大于等于 50%,有一定的缺陷检测能力. 本实验中 4-gram 模型较 3-gram 模型和 5-gram 模型缺陷检测效果较好,但在本实验中它们均不如基准模型(查准率

最高和基准持平）.而相较而言, DeepLSTM 模型在本次实验所采用的模型中有一定的检测能力且效果最好.

当可接受阈值取到 Top-8 时,DeepLSTM 缺陷检测效果最好.由此可以看出在本实验中的代码缺陷模型的 Top-8.

为进一步探究 API 误用类型与模型能力的关系,实验中,将 DeepLSTM 模型对应每个可接受阈值（Top-k）的具体 API 误用缺陷报告情况整理在表 2 中,表格中检测正确的 API 误用用“√”符号表示,其余则记为未检出的 API 误用.

Table 2 API misuse bug detection reports statistics

表 2 API 误用缺陷检测报告情况统计

测试用例	Top-1	Top-2	Top-3	Top-4	Top-5	Top-6	Top-7	Top-8	Top-9	Top-10
1	√									
2	√	√	√	√	√	√	√	√		
3	√	√	√	√	√	√	√	√	√	√
4	√									
5	√	√	√	√	√	√	√	√	√	√
6	√	√	√	√	√	√	√	√	√	√
7	√	√	√	√	√	√	√	√	√	√
8	√	√	√	√	√	√	√	√	√	√
9	√	√	√	√	√	√	√	√	√	√
10	√	√	√	√						
11	√	√	√	√	√	√	√	√	√	√
12										
13	√	√	√	√	√	√	√	√	√	√
14										

基于整理后的 API 误用缺陷检测报告情况,按照测试集中 API 误用的误用类型对缺陷检测报告情况进行进一步的可视化统计分析,如图 10 所示.

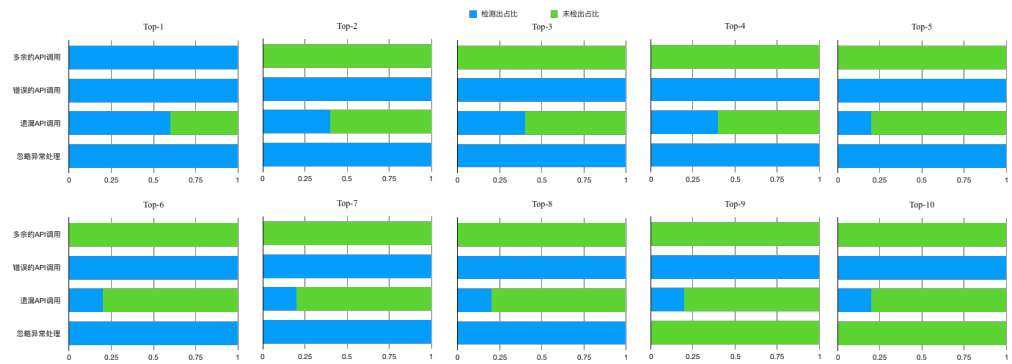


Fig.10 Analysis of relationship between acceptable threshold and type of API misuse

图 10 可接受阈值与 API 误用类型关系分析

纵轴表示了不同类型的 API 误用缺陷,横轴表示检测出以及未检出的该类型 API 误用数量在该类型 API 误用总数中的占比,用颜色深蓝色、浅绿色分别表示检测出和未检出的两种情况,图中有十个独立的子图来分别表

示每个可接受阈值（Top-k）对应的 API 误用缺陷检测报告情况.

分析该图,可以看出：

随着 k 取值的增大,在某些 API 误用的检测上,模型效果保持基本不变（错误的 API 调用）,一定程度说明了本实验中采用的缺陷检测方法对检测该类 API 误用的有效性：当一个 API 误用在 API 调用序列中不应该出现（使用了错误的 API）,则应用本实验中按位置预测并比较检测 API 误用是可行且有效的;

但是与之相反的,随着 k 取值的增大,在某些 API 误用的检测上,模型效果不断降低（遗漏 API 调用）,这一定程度上也说明了本实验中采用的缺陷检测方法对检测该类 API 误用的局限性：当一个关键性 API 在 API 调用序列中遗漏,仅靠当前的 API 调用序列作为预测的前文,很难推断出下一位置是否遗漏了某个 API,例如,某个关键性 API 可能在当前位置更靠后几个调用中出现,这种情况对目前采用的仅对一个位置的 API 误用缺陷进行检测的方法就具有很大的挑战性;

随着 k 取值的增大,在某些 API 误用的检测上,模型效果有较大的突变（多余的 API 调用、忽略异常处理）,这既与本实验中采用的测试集有一定的关系——多余的 API 调用和忽略异常处理的测试用例都仅有一个,但是在 k 取到一定范围（Top-8）之前,本实验中采用的缺陷检测方法对忽略异常处理类型的 API 误用缺陷检测效果还是不错的;

就每个独立子图而言,也可以验证以上几点结论：本实验采用的 API 误用缺陷检测方法,对检测错误的 API 调用类型的 API 误用效果较好,而在发现遗漏 API 调用类型的 API 误用上,能力稍有欠缺.

综上,本实验应用深度学习模型学习大量代码中的 API 使用规约,并应用在 API 误用缺陷检测上的方法.虽然由于大量开源代码中可能存在些许误用的 API 代码,影响到训练数据的质量,对模型的效果产生着一定的影响,但是本实验中模型在以检测 Java 加密相关的 API 误用代码缺陷为例的实验中仍然起到了一定的效果,以可接受阈值为 8 时效果最佳（F1 值、准确率）.在本实验中,模型对检测错误的 API 调用类型的 API 误用效果较好,而在发现遗漏 API 调用类型的 API 误用上,能力稍有欠缺,这与每种 API 误用类型内在的规律特征密不可分.

6 结束语

本文将深度学习中的循环神经网络模型应用于 API 使用规约的学习及 API 误用缺陷的检测.将 API 调用组合、顺序及控制结构等方面的使用规约建模为 API 语法图和 API 调用序列.在大量的开源 Java 代码基础上,对代码进行静态分析,并构造大量 API 使用规约训练样本,对循环神经网络进行训练.在实验中通过尝试不同参数组合,选定并训练出较优的循环神经网络模型,并用于基于前文的 API 调用预测,通过预测结果与实际代码进行比较来发现潜在的 API 误用缺陷.随后,在以检测 Java 加密相关的 API 误用代码缺陷为例的实验中证明了该方法的有效性,实验表明,本方法检测错误的 API 调用类型的 API 误用时,最为稳定有效.

本文提出的方法在一定程度上能自动检测 API 误用缺陷,并在某类 API 误用的检测中较为有效,但是还存在一些不足和可改进之处——在发现遗漏 API 调用类型的 API 误用上,能力稍有欠缺,在未来研究中,可考虑对多个位置的 API 调用进行预测比对,减少模型由于单个位置的预测带来的 API 是否漏用的不确定性.

References:

[1] Li Z, Wu JZ, Li MS. Study on key issues about API usage. Ruan Jian Xue Bao/Journal of Software, 2018,29(6).

[2] Zhou Y, Gu R, Chen T, Huang Z, Panichella S, Gall H. Analyzing APIs documentation and code to detect directive defects. In: Proc. of the 39th Int'l Conf. on Software Engineering. Piscataway: IEEE Press, 2017. 27−37.

[3] Liu S, Bai G, Sun J, et al. Towards Using Concurrent Java API Correctly[C]. In: Proc. of the 21st Int'l Conf. on Engineering of Complex Computer Systems (ICECCS), Dubai, 2016, pp. 219-222.

[4] Sacramento P, Cabral B, Marques P. Unchecked exceptions: can the programmer be trusted to document exceptions[C]. In Second International Conference on Innovative Views of .NET Technologies, Florianópolis, Brazil. 2006.

[5] Amann S, Nadi S, Nguyen H A, et al. MUBench: a benchmark for API-misuse detectors[C]. In: Proc. of the 13th Int'l Conf. on Mining Software Repositories. ACM, 2016: 464-467.

[6] Gao Q, Zhang H, Wang J, et al. Fixing recurring crash bugs via analyzing q&a sites (T)[C]. In: Proc. of the 2015 30th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE), IEEE, 2015: 307-318.

[7] Zhong H, Zhang L, Mei H. Mining invocation specifications for API libraries. Journal of Software, 2011,22(3): 408−416. http://www.jos.org.cn/1000-9825/3931.htm

[8] Li Z, Zhou Y. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. SIGSOFT Software Engineering Notes, 2005,30(5):306−315.

[9] Zhong H, Zhang L, Xie T, Mei H. Inferring resource specifications from natural language API documentation. In: Proc. of the 2009 IEEE/ACM Int'l Conf. on Automated Software Engineering. Washington: IEEE Computer Society, 2009. 307−318.

[10] Wang S, Chollak D, Movshovitz-Attias D and Tan L. Bugram: Bug detection with n-gram language models. In: Proc. of the 2016 31st IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE), Singapore, 2016, pp. 708-719.

[11] website: https://en.wikipedia.org/wiki/Deep_learning

[12] Zheng ZY, Liang BW, Gu SY. TensorFlow: Google deep learning framework, put it into practice (2nd ed.) [M]. Beijing: Publishing House of Electronic Industry,2018(in Chinese)

[13] Graves A, Jaitly N, Mohamed A. Hybrid speech recognition with deep bidirectional LSTM[C]. In 2013 IEEE Workshop on Automatic Speech Recognition and Understanding. IEEE, 2013: 273-278.

[14] Luong M T, Pham H, Manning C D. Effective approaches to attention-based neural machine translation[J]. arXiv preprint arXiv:1508.04025, 2015.

[15] Cho K, Van Merriënboer B, Gulcehre C, et al. Learning phrase representations using RNN encoder-decoder for statistical machine translation[J]. arXiv preprint arXiv:1406.1078, 2014.

[16] Olah C, Understanding LSTM Networks, [EB/OL],2015. [http://colah.github.io/posts/2015-08-Understanding-LSTMs/#fn1]

[17] Hochreiter S, Schmidhuber J. Long short-term memory[J]. Neural computation, 1997, 9(8): 1735-1780.

[18] LI H. Statistical Learning Method [M]. Beijing: Tsinghua University Press, 2012. (in Chinese)

[19] Smith N, Bruggen D V and Tomassetti F. JavaParser: Visited Analyse, transform and generate your Java code base, [EB/OL], 2017. [https://enterprise.leanpub.com/javaparservisited]

[20] Grinberg M. Flask Web Development[J]. Oreilly Vlg Gmbh & Co, 2014.

[21] Nadi S, Kr S, Mezini M, Bodden E. Jumping through hoops: Why do Java developers struggle with cryptography APIs? In: Proc. of the 38th Int'l Conf. on Software Engineering. New York: ACM Press, 2016. 935−946.

[22] Egele M, Brumley D, Fratantonio Y, and Kruegel C. An empirical study of cryptographic misuse in Android applications. In Proc. of the Conference on Computer and Communications Security (CCS), pages 73–84, 2013.

[23] Fahl S, Harbach M, Muders T, Smith M, Baumg¨artner L, and Freisleben B. Why Eve and Mallory love Android: An analysis of android SSL (in)security. In Proc. of the Conference on Computer and Communications Security (CCS), pages 50–61, 2012.

[24] Gousios G, Spinellis D. Mining software engineering data from GitHub[C]. In: Proc. of the 2017 IEEE/ACM 39th Int'l Conf. on Software Engineering Companion (ICSE-C). IEEE, 2017: 501-502.

[25] Fowkes J, Sutton C. Parameter-free probabilistic API mining across GitHub[C]. In: Proc. of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2016: 254-265.

[26] Stanley F. Chen and Joshua Goodman. 1996. An Empirical Study of Smoothing Techniques for Language Modeling. In Proceedings of the 34th Annual Meeting on Association for Computational Linguistics (ACL'96). Association for Computational Linguistics, Stroudsburg, PA, USA, 310–318.

附中文参考文献:

[1] 李正,吴敬征,李明树.API 使用的关键问题研究. 软件学报,2018,29(6):1716-1738.

[7] 钟浩, 张路, 梅宏. 软件库调用规约挖掘[J]. 软件学报, 2011, 3: 003.

[12] 郑泽宇,梁博文,顾思宇.TensorFlow: 实战 Google 深度学习框架（第二版）[M]. 电子工业出版社,2018

[18] 李航. 统计学习方法[M]. 清华大学出版社, 2012.