软件标识符的自然语言规范性研究

汶东震，张帆，张晓，杨亮，林原，徐博，林鸿飞†*
大连理工大学/辽宁省大连市
sewen@mail.dlut.edu.cn, fanz@mail.dlut.edu.cn, kun@mail.dlut.edu.cn
liang@dlut.edu.cn, zhlin@dlut.edu.cn, xubo@dlut.edu.cn
hflin@dlut.edu.cn

摘要

软件源代码的理解则是软件协同开发与维护的核心，而源代码中占半数以上的标识符的理解则在软件理解中起到重要作用，传统软件工程主要研究通过命名规范限制标识符的命名过程以构造更易理解和交流的标识符。本文则在梳理分析常见编程语言命名规范的基础上，提出一种全新的标识符可理解性评价标准。具体而言，本文首先总结梳理了常见主流编程语言中的命名规范并类比自然语言语素概念本文提出基于软件语素的标识符构成过程，即标识符的构成可被视为软件语素的生成、排列和连接过程。在此基础上，本文提出一种结合自然语料库的软件标识符规范性评价方法，用来衡量软件标识符是否易于理解。最后，本文通过源代码理解数据集和Github平台中开源项目对规范性指标进行了验证性实验，结果表明本文提出的规范性分数能够很好衡量软件项目的可理解性。

关键词： 软件标识符 ； 源代码理解 ； 软件维护 ； 自然语言模型

Research on the Natural Language Normalness of Software Identifiers

Dongzhen Wen, Fan Zhang, Xiaokun Zhang, Liang Yang, Yuan Lin, Bo Xu, Hongfei Lin
sewen@mail.dlut.edu.cn, fanz@mail.dlut.edu.cn, kun@mail.dlut.edu.cn
liang@dlut.edu.cn, zhlin@dlut.edu.cn, xubo@dlut.edu.cn
hflin@dlut.edu.cn

Abstract

The understanding of identifiers plays an important role in software understanding. In this paper, we propose a new criterion for evaluating the comprehension of identifiers. Firstly, we compare the naming conventions in mainstream programming languages and propose a Software Morpheme-based identifier composition process. Specifically, identifiers can be considered as an arrangement and concatenation of different software morphemes. On these basis, this paper proposes a new evaluation metric for software identifier understandability named the normalness of identifiers. Finally, this paper conducts experiments on the normalness metric through source code comprehension tasks and open source projects on the Github platform. The results show that the normalness scores we proposed can measure the understandability of software projects.

Keywords: software identifiers , source code understanding , software maintenance , natural language models

第二十届中国计算语言学大会论文集，第837页-第850页，呼和浩特，中国，2021年8月13日至15日。

1 引言

软件已经成为信息社会发展的基石，随着软件需求的日益增多，软件开发与维护也变得日益困难。早期单人独立成军（one man army）的开发模式在大规模软件系统开发时已经变得并不适用，随后一些软件开发实践中尝试通过堆砌人力来解决开发问题。但实践证明，单纯堆砌开发人员往往不能实现有效的开发效率提升效果，甚至会适得其反(Frederick, 1995)。其主要原因在于，开发者需要在理解软件代码的基础上对源代码进行增补和修复。这种软件理解（program comprehension）的需求往往随着团队规模的增大而呈几何倍率增大。

其中软件结构复杂性(McCabe, 1976)以及编程风格是影响软件理解的重要因素。软件结构复杂性方面，源代码中的分支结构、循环结构影响着开发者对于代码局部功能的认知，而现代软件开发过程中遵循的设计模式（design pattern）则使得开发者不得不在多个文件之间反复跳转从而理解代码功能。而编程风格方面，代码标识符设计对于软件理解起着重要作用(Lawrie et al., 2007a)。好的标识符命名能够向阅读者提供软代码更丰富的信息，降低开发者对于程序结构理解和推断的阅读负担，从而提高开发者对于代码理解的效率。

从软件组成上来看，软件源代码由标识符（Identifiers）、操作符（Operators）以及编程语言特定的关键字（Keywords）组成，其中开发人员可自由命名的标识符占比最大，约为70%(Jiang et al., 2020)。软件标识符是一种由表示对象、动作的名词、动词以及其他连词以及连接符号构成的表示源代码单元含义的注解符号。Carter(Carter, 1982)的研究中戏称开发者在软件开发协作时不仅仅在学习一门全新的语言（理解他人书写的标识符），同时他们自身也在花费大量时间为现有的事物”创造“新的名称。软件开发与维护过程中，开发者往往需要在理解现有项目代码基础上进行开发与维护，软件源代码的可理解性尤其是标识符的可理解性在其中至关重要。

于此同时，在软件项目（Project）、模块（Module）、类（Class）、方法（Method）等不同粒度，源代码都具备较强的重复性和局部性特点(Tu et al., 2014)。如：在项目级别软件编程规范的前后一致性，模块级别命名规范的一致性，类和方法中相同标识符的局部重复性和语义确定性等等。在此基础上，Hindle等人(Hindle et al., 2012)提出软件的自然性假设，即：

> **软件作为一种人类创造的交流方式，软件语料库具有与自然语言语料库相似的统计特性。我们可以利用这类统计规律来构建更好的软件工程辅助工具。**

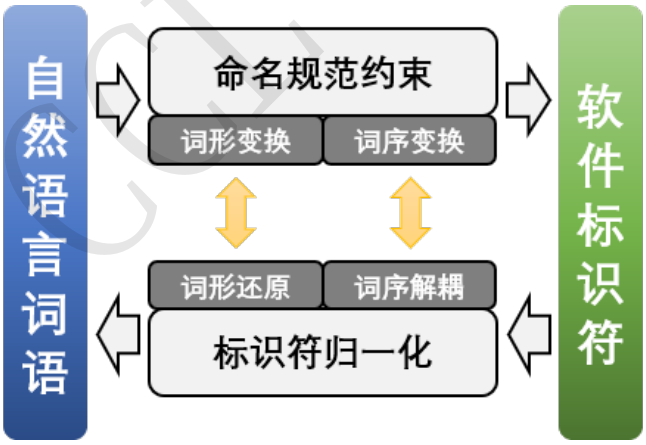

Figure 1: 标识符的构成和理解

软件的自然性假设使得我们能够将自然语言理解模型应用在软件源代码数据中，从而构建提高开发者对代码理解能力的工具。基于自然语言表示模型的源代码表示任务，基于信息检索的代码搜索任务和推荐任务、基于文本摘要技术的代码摘要和注释文档生成任务等纷纷被提出，用以指导源代码理解研究。其中软件标识符作为由自然语言词语构成的源代码成分是软件源代码理解和自然语言理解之间的桥梁。

实际代码阅读过程中，代码标识符命名的任意性往往导致代码理解困难(Carter, 1982)。因此，在软件开发实践中，主要通过标识符命名规范和归一化两种方式减弱由标识符带来的理解困难。标识符命名规范（Naming Conventions）约束主要开展于软件项目启动之前。团队首先根据开发需求统一代码编写过程中所用单词词表，在此基础上约定词组的简写和排列方式。这种事前制定的命名规范对于标识符的可理解性有着重要影响(Binkley et al., 2013a)。除此之外，研究者在对大量软件项目数据进行分析基础上，构建词典、启发性规则以及机器学习模型，针对软件项目中标识符进行归一化（Normalization）分析。这类工具往往能够将标识符还原为表达明确含义的自然语言短语，从而为开发者理解标识符具体含义提供参考(张静宣;江贺;, 2020)。命名规范和标识符归一化研究从软件开发的两个不同阶段来对标识符可理解性进行明确。如图1所示，自然语言词组通过一定命名规范约束转化为软件标识符，反之标识符通过归一化方法被重新解读为自然语言词组。标识符命名规范原则往往能够作为启发式规则被用于标识符到自然语言的还原，同时结合统计规律对标识符进行还原时相关结论会形成新的命名规范。

通过对命名规范的研究，本文将开发者构建软件标识符的过程看作软件词素（Software Morpheme）的形变和排列连接两个主要过程。具体而言，编码过程中开发者首先根据需求确定标识符语义，同时对词形进行简洁化构成软件词素，不同软件词素之间的排列和连接构成软件标识符。在此基础上，本文提出软件标识符的自然语言规范性（Normalness）指标，用来衡量软件标识符的可理解性。

本文对构成标识符软件词素词形、不同词素之间的词序和词组合特性三个方面进行评价。结合自然语料库研究软件词素和自然语言词语之间分布的一致性并构建标识符规范性指标。实验部分，本文分别从人类可理解性和模型可理解性两个方面对规范性分数评价能力进行验证。

人类可理解性方面，本文从Github开源平台采集不同协作程度的开源项目数据，对规范性分数和协作性之间相关性进行分析。模型可理解性方面，本文在实际软件理解任务数据集上研究规范性分数和自然语言模型在软件理解任务中适应性之间的关联。结果表明，本文提出的标识符自然语言规范性指标能够很好衡量命名规范在自然语言模型上的应用。

本文剩余部分组织方式如下：第2章总结整理本文研究基础以及相关工作；第3章描述本文规范性评价过程；第4章验证规范性指标在实际项目以及代码理解任务上的评价能力；第5章总结全文并提出未来研究的展望。

2 研究基础

在软件项目开发过程中，为保证编程风格的一致性以及项目整体的协作性，软件开发团队往往会提前约定相应的命名规范(Wikipedia contributors, 2021)（naming conventions）对项目开发过程中标识符构建过程进行约束。Aggarwal等人(Aggarwal et al., 2002)研究表明，软件源代码的可读性和文档可读性在软件维护中有着相同的重要性。标识符的可理解性对于代码理解有着重要影响，软件工程实践中主要使用命名规范对标识符的构造进行约束。典型的命名规范如表2所示，可以看到标识符构建时主要考虑用词和词之间的连接问题。根据连接方式不同，可以分为大写字母分隔和下划线分隔两种，而连字符命名方法只在Lisp类语言中适用。而根据单个单词首字母的不同，命名规范可进一步分为大驼峰命名法（帕斯卡命名法）小驼峰命名法等。

而在开发实践中，不同编程语言对于命名规范的选择也各有侧重。Java(Google, 2021b)、Javascript(Google, 2021c)、Go(Golang, 2021)、Php(pear, 2021)、.net(Microsoft, 2021)等语言倾向大量使用驼峰命名法作为编程规范中的指导意见。而Python(Google, 2021d)、C/C++(Google, 2021a)和Ruby(rubocop, 2021)等语言则倾向于使用下划线命名风格对成员和方法进行命名。尽管不同语言存在一定差异，但各类项目命名规范都遵循以下原则(Google,)：

（1）构成标识符所使用词语应当表意明确且形式简洁，其中明确性要求大于简洁性要求；

（2）构成标识符的词语应当尽可能避免歧义，避免使用容易引起混淆的缩略词；

（3）词语之间需要能够以清晰明确的方式进行连接，特定情况下允许使用反模式使用；

Butler等人(Butler et al., 2010a)收集了多个Java项目，并在此基础上经验性地研究了标识符质量和软件质量之间的关系。Binkley等人(Binkley et al., 2013b)设置五项不同实验，针对不同标识符风格对于代码理解的影响进行研究。作者基于阅读时间测试，大声朗读测试和眼动追踪等方法对人类阅读不同标识符的认知过程进行记录。结果表明，源代码阅读理解过程与

命名法	定界符	首词	其他词	示例	对应实践
小驼峰命名法	大写字母	全小写	首字母大写	camelCase	Java; JS;Php;
帕斯卡命名法	大写字母	首字母大写	首字母大写	PascalCase	Go; .net;
蛇形命名法	下划线	全小写	全小写	tree_child_prt	Py; C; Ruby;
常量命名法	下划线	全大写	全大写	RAND_SEED	常量
烤肉串命名法	连字符	无要求	无要求	KeBabe-Case	Lisp
交替大写命名法		反模式		sTuDlYcApS	反模式
匈牙利命名法	大写字母	全小写	首字母大写	lVerticalSide	MFC

Table 1: 常用命名规范原则及示例

自然语言阅读理解有着很大的不同。经验丰富的软件开发人员倾向于更少受标识符样式的影响。而但从准确性和费力程度来讲，驼峰命名法更加适合初学者阅读。Liblit等人(Liblit et al., 2006)研究表明，在标识符命名过程中，影响其可理解性的两大因素分别为简洁性和一致性。标识符过于冗长会加重阅读者的认知负担，而一致性较差则会带来理解混淆的问题。在其(Lawrie et al., 2007b)另一项研究中发现，程序员倾向于使用相当有限的词汇构建标识符。但在构造标识符词语时或多或少会违反句法规则，这一现象在专有软件项目和开源软件项目中比例各有不同。Ebad等人(Ebad and Manzoor, 2016)收集了120个Java和C#项目，对项目中标识符对命名规范的符合情况进行了评估。研究表明，违反命名规范的情况往往和类的大小以及项目的规模有着相关关系。往往随着项目和单个类的增大，违反命名规范的情况会逐渐增多。

Borstler等人(Börstler et al., 2016)研究发现，标识符长度对于开发者阅读源代码时的认知难易程度有着重要影响。基于这一结论，作者结合平均句子长度和平均单词长度提出软件简单可读性度量指标SRES。进一步研究表明，SRES指标在学生的教学过程中有着良好表现。Raymond等人(Buse and Weimer, 2010a)针对代码可读性的概念提出一种自动化可读性度量方案。作者针对源代码构建了多组特征，并结合分类模型对可读性指标进行评价。研究结果表明，过多注释往往会降低代码可读性，使用空白行对代码进行分隔反而优于添加注释。

研究者在标识符可理解性基础上，提出标识符的归一化研究，旨在将标识符转化为规范的自然语言词组。江贺等人(张静宣;江贺;, 2020)针对软件标识符归一化研究进行了总结梳理。其将软件标识符归一化总结为组合词拆分以及缩写词扩充两个阶段，结合启发式算法以及现有的英语词典和缩略词词典，对标识符中的缩略词理解问题进行解决。除缩略词之外，软件标识符中的同义词(Pirapuraj and Perera, 2017)、近义词(Arnaoudova et al., 2014)等也会对源代码理解产生较大影响。Wang等人(Wang et al., 2014)研究则表明，开发人员在创建标识符名称时使用了多种语法和形态学手段，在遵循一定命名规范的同时也会创造新的命名规范。进而促进命名规范和标识符归一化研究。

可以看到，传统的软件工程研究主要结合案例分析、调查问卷、实证研究以及统计模型等方法对代码可理解性进行评估。而近期，越来越多的软件源代码理解任务和数据集被提出，旨在通过机器学习手段深入进行源代码理解研究。常见任务包含代码搜索(Husain et al., 2019; Gu et al., 2018)、代码摘要生成(Zhu and Pan, 2019)、基于检索的软件缺陷定位(Lawrie and Binkley, 2018)、软件缺陷报告摘要(Rastkar et al., 2014; Li et al., 2018)以及代码克隆检测(White et al., 2016)等任务。

本文则结合软件源代码理解任务和开源项目案例分析两种手段，对源代码可理解性进行评估。

3 软件语素与标识符规范性

可以看到，软件标识符作为自然语言词语（主要为英文）构成的词组短语，其可读性主要取决于两点：构成标识符的词语是否常见以及词组之间的排列方式是否符合自然语言常见表达方式。基于以上假设，本节首先对软件标识符的构成过程进行梳理，在此基础上，结合构成标识符的词形和词的排列特性，提出软件标识符的自然语言规范性评价方法。

3.1 软件语素与标识符

软件标识符作为领域特定的词组，其构成方式与自然语言词和词组的构成方式类似。不同

的命名规范通过约束标识符构成过程中词选择和组合过程来确保标识符表意清晰便于阅读和理解。本文中将构成标识符的基本表意单元定义为**软件词素**。即能够独立表示软件上下文中特定含义，承担表达软件开发相关知识的词称之为软件词素。软件词素主要包括自然语言词、自然语言词的简写形式以及软件开发中约定俗称的缩略词三种形式。在此基础上，代表开发活动具体含义的软件词素按照命名习惯或自然语言表达习惯进行排列和连接，最终构成软件标识符。整体流程如图2所示。

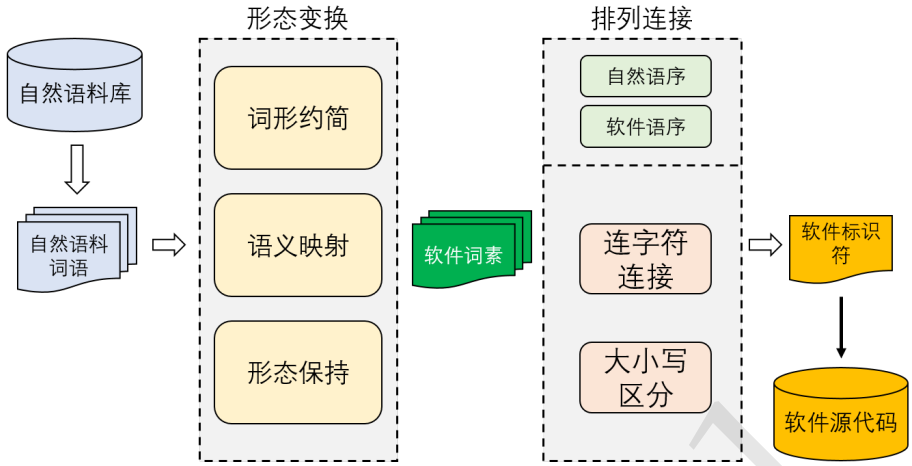

Figure 2: 软件标识符的构成过程

此处，设自然语言语料库为C_n，自然语言词组t为语料库中的词组，$t \in C_N$。构成标识符的自然语言词集合$\{t_1, t_2, ..., t_p\}$经过形态转换过程后得到软件词素$\{m_1, m_2, ...m_p\}$，即：

$$\{m_1, m_2, ...m_p\} = Transform(\{t_1, t_2, ..., t_p\}) \tag{1}$$

在此基础上，软件词素通过特定排列方式得到标识符候选序列。此处软件语素的连接方式包括符合自然语言表达规范的自然语序，以及和具体软件开发任务相关的软件特定语序两种形式。

$$[m_1, m_2, ..., m_p] = Arrange(\{m_1, m_2, ...m_p\}) \tag{2}$$

最后，软件词素候选序列会根据标识符自身角色，结合特定编程语言命名规范，使用连字符$< Sep >$进行语素的连接，构成标识符词组。

$$Identifier = Concat([m_1, m_2, ..., m_p], < Sep >) \tag{3}$$

一些标识符的构成实例如表2所示，包含原始词语、软件语素以及通过一定连接方式构成的标识符词例，与图2描述过程一致。从自然语言词组出发，不同词通过形态保持、约简以及词义映射等方法，形成软件语素，构成标识符的语素按照自然语言表达规范以及编程常见规范进行排列，最后通过连字符或者大小写区分的方式相互连接，构成标识符。

变换方法	自然语言词（组）	软件语素	标识符例
形态保持	open; load; adapter	open; windows; adapter	openFileDialogue;
形态约简	Kubernetes; pointer; deep-first search	k8s; ptr; DFS	k8s_token;
词义映射	serialization; deserialization	dump; load	JsonDumpAdapter

Table 2: 标识符构成示例

具体而言，形态约简和词义映射分别考虑标识符构造中软件语素的简洁性和表意明确性。

形态约简变换中，常用方法为首字母缩略词（Acronym），即将词组中多个单词的首字母保留，构成缩略词形式。一些计算机常见术语也已经被编纂为词典便于快速查阅[0]。除此之

[0]https://techterms.com/category/acronyms

外，许多编程实践中建议从单词首字母开始保留一段连续长度的字母序列作为单词的缩略形态(Carter, 1982)。但在具体实践中，如pointer（指针）这一单词，使用截取方法相对无法进一步缩略，因此去掉元音字母后的'ptr'更加适合作为其缩略形态。而容器编排平台Kubernetes，往往被工程师缩写为'k8s'，取其首位字母以及中间的字母个数来拼凑称为原始词的缩略形态。

而词义映射则主要考虑软件语素的表意明确性问题，即构成标识符的表意单元是否足够清晰传达作者意图，降低开发者在阅读的认知负担。因此词义映射部分通过同义、近义词（synonym）等，将较为复杂词语转换为更为清晰的形式。如表2中所示，序列化和反序列化操作在软件开发过程中十分常见，但作为软件词素构成标识符时较为冗长，其缩略词形式'serial'和'deserial'表意不明，在阅读时认知负担较重。因此往往采用'dump'和'load'等，表示相近含义的词进行替代，确保软件词素的表意明确。

可以看到，标识符构成过程中针对词的简洁性和表意表意明确性之间做出较多权衡，包括词的形态变换以及语义变换（映射）均会对代码的可理解性产生较大影响。下文中将着重对词形词序对于代码可理解性的影响进行评估。

3.2 标识符规范性评价

上节中针对标识符的构造过程进行了描述，可以看到，构建表示特定含义的标识符主要经过自然语言表意词选择、软件语素转换以及语素排列三个过程。相应的，对构成标识符的自然语言词组的还原过程也主要分为标识符切分和软件语素转换两个过程(张静宣;江贺;, 2020)。本节则针对软件语素的排列特性，提出标识符规范性评价方法，结合标识符归一化方法对标识符可理解性进行分析。

评价对象

标识符规范性评价对象方面，主要从自然语言构词角度，评价软件语素中三个方面规范性情况：

- 词形规范性：评价软件语素是否符合自然语言表达规范，即是否存在新词、造词等现象。
- 词组规范性：评价软件语素之间的组合（共现）情况是否符合自然语言表达规范，是否存在生僻的词组组合情况。
- 词序规范性：评价软件语素之间的排列规范性，在词组合评价基础上，评价构造标识符的短语是否符合自然语言表达规范。

具体而言，我们结合N元语法(Suen, 1979)和词共现方法进行评价。如表3.2所示，我们使用标识符对应软件语素的二元语法和三元语法形式以及同一标识符对应的软件语素之间的组合特性作为评估对象。

标识符例	软件语素	词形规范性	组合规范性	词序规范性
AbC	A; b ;C	AbC	{A,b}; {b,C}; {A,C}; {A,b,C}	A_b; b_C; A_b_C

Table 3: 规范性评价对象示例

特别的，Raymond等人研究表明(Buse and Weimer, 2010b)过长的标识符也会带来开发者的阅读负担。因此本文构建标识符规范性评估方法时选择软件语素的原始形式，以及其二元语法和三元语法形式，四元语法以及更长的形式则不予考虑。

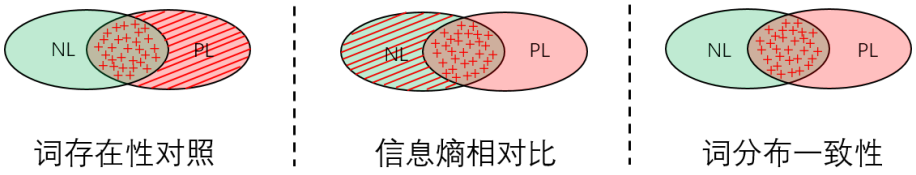

Figure 3: 三种评价维度

评价维度

评价维度方面，本文则基于自然语料库，分别从词的存在性、信息熵相对比以及分布一致性三个维度对软件语素对自然语言的规范性程度进行评价。

三种比对过程如图3所示。其中三者左侧绿色集合代表自然语料库，右侧红色集合代表具体一个软件项目中所有标识符经过归一化之后得到软件语素的集合。两个集合交集部分表示二者共享的词语或词组集合，即形态规范的软件语素。

本文记自然语言语料库为C_N，具体一个软件项目所有标识符归一化后构成的软件语素语料库为C_P，两个语料库交集S_I。此处我们注意到自然语料库C_N和软件语素语料库C_P中包含词组及其对应的词频，其在词空间上构成离散分布。记$t_n \in C_N$为自然语料库中词组，$t_p \in C_P$为软件语素。则记$P_N(t_n)$和$P_P(t_p)$分别为词t_n和t_p在两个语料库上的分布频率（当前词的频次除以所有词总频次）。

词存在性计算：如图3左侧部分所示。词存在性主要评估规范的软件语素占构成标识符的软件语素总体的比值。此处记软件语素和自然语料库之间的交集S_I为规范的软件语素，则词存在性分数可记为：

$$Existence = \frac{\|S_I\|}{\|C_P\|} = 1 - \frac{\|C_{oov}\|}{\|C_P\|} \tag{4}$$

可以看到词的存在性分数同时可以视为1与软件语素中对自然预料库的集合外词（out of vocabulary）C_{oov}占比之差。

信息熵相对比计算：此处本文结合信息熵概念对软件语素中自然语言规范性进行分数进行衡量。我们以自然语料库C_N信息熵作为对比，计算构成标识符的软件语素信息熵大小和自然语料信息熵之比。具体计算过程如下：

$$EntropyRatio = \frac{\sum_{t_p \in C_I} P_N(t_p) * \log(P_N(t_p))}{\sum_{t_n \in C_N} P_N(t_n) * \log(P_N(t_n))} \tag{5}$$

如图3中间部分所示，分子部分从规范的软件语素集合中取词t_p计算其在自然语素语料库上的分布概率，乘以取对数的概率并求和作为规范软件语素的信息熵。分母部分为自然语料库整体的信息熵，计算过程和分子部分一样，当对照用的语料库确定时，分母为固定值。相比于词存在性分数，信息熵相对比在衡量规范的软件语素在自然语料库中占比的同时，将词的信息量考虑在内，更充分衡量软件语素相对自然语料库的规范性情况。

词分布一致性计算：此处本文使用KL散度（Kullback-Leibler divergence即相对熵）对软件语素相对自然语料库的分布一致性进行评价。计算过程如下：

$$\begin{aligned} KLD_{(N,P)} &= \sum_{t_n \in C_I} P_N(t_n) \log(\frac{P_N(t_n)}{P_P(t_n)}) \\ Consistency &= e^{-KLD_{(N,P)}} \end{aligned} \tag{6}$$

如图3所示，分布一致性分数计算中，主要针对规范的软件语素分布进行比较。其中自然语言语料库为规范分布P_N，而软件语素分布为对照分布P_P。相对熵分数越大说明软件语素与自然语言词组分布之间的分布差异越大，从分布角度讲规范性相对更低。为方便分数计算，此处我们取相对熵相反数的自然指数形式，将值域放缩到0到1之间。

$$Normalness = Existence + EntropyRatio + Consistency \tag{7}$$

指标融合方面，本文参考了(Butler et al., 2010b; Buse and Weimer, 2010c)等工作。传统软件工程对代码可读性评价时往往引入大量经验参数构造打分标准，而通过机器学习方法进行参数学习又存在结论泛化性问题。因此本文在前人工作基础上简化了指标融合方式，通过直接加和三项指标的方式融合得到标识符规范性评价指标7。

4 实验结果评估

上文中针对软件标识符的自然语言规范性提出了具体的评价指标，本节则针对这一指标的有效性进行评价。我们首先提出假设，源代码中标识符规范性越高相应的其可理解性也会越强。本节从模型可理解和人可理解两个方面对规范性指标进行评价。其中模型可理解性方面，本文通过不同自然语言模型在代码搜索任务上的效果情况，对比不同规范性指标下代码搜索任务表现。二者成正向相关则说明规范性指标对于代码的模型可理解性有较好的反映。

而人可理解性方面，本文则收集github平台上开源项目和项目相关元信息，评估项目受欢迎程度和协作广度程度和规范性指标之间的关系。其中受欢迎程度主要通过开源项目的收藏（star）数目进行评估，而协作广泛程度则通过开源项目的关注（watch）数目以及分支（fork）数目来具体衡量。本文认为受欢迎且协作广泛的软件项目其代码相对更易理解。相应的标识符规范性指标相也会更高，同时协作广泛性和受欢迎程度与规范性指标呈现正相关。

为了确保规范性评价时的合理性，本文中用于对照源代码规范性的语料库使用维基百科[1]数据。本文下载了截止2021年5月的英文维基百科数据，通过数据清洗去除其中链接、符号以及其他无关元字符后，构建英文维基百科词表并统计词分布情况，作为源代码标识符规范性评价的自然语言词分布参照。

4.1 实验设置

方法	示例	软件语素	评价对象
全小写处理	JsonDumper	jsondumper	词形规范性
下划线分隔	event_collector	event; collector	词形、词组合、词义规范性
大写字母分隔	NetworkPolicy	network; policy	词形、词组合、词义规范性

Table 4: 归一化方法和评估对象

结合上文对于规范性的讨论可以看到，规范性评价对象主要为软件语素单元。标识符首先需要通过归一化方法转化为软件语素后再进行评估。为研究方便，本文使用最常见的两种标识符归一化方法对代码文本进行处理，评估的方法和维度具体如表所示4。下划线分词方法和大写字母分割法进行标识符归一化，切分之后所有软件语素被转换为全部小写形式以便于分析统计。通过下划线和大写字母对标识符进行切分后，可以进行软件语素之间的共现关系和排列顺序情况进行评价。同时，为保证对比合理性，本文将全小写化的标识符作为一种标识符归一化方法进行结果对比，主要侧重针对软件语素词形规范性进行评价。

代码理解任务评估

编程语言	Go	Java	Javascript	Python	Php	Ruby
训练集	317832	454451	123889	412178	523712	48791
验证集	14242	15328	8253	23107	26015	2209
测试集	14291	26909	6483	22176	28391	2279
标注数据	166	813	1819	2079	314	315

Table 5: CodeSearchNet数据集情况

本文使用微软发布的CodeSearchNet[2]数据集作为评估数据集。数据集包含来源于Github的实际项目代码，涵盖六种编程语言。数据集主要形式为代码片段–注释文本对，以供研究者构建语义匹配模型建模源代码和自然语言之间语义关系。评估数据标注部分主要包含99条自然语言查询文本，每条文本对应在六种编程语言的数据集上进行相关性标注，标签等级涵盖0,1,2,3四个级别，以衡量查询和代码片段之间的相关程度。在此标注基础上，我们主要通过归一化累计增益分数（NDCG）作为搜索效果的评价指标。

检索模型部分如图4所示，本文针对Gu等人(Gu et al., 2018)的深度代码搜索架构进行了改进，代码片段和文本片段通过语言模型获得表示向量之后分别通过两个相同结构的编码器获得表示向量。最后通过特征交互层得到代码片段和对应文本之间的相关性分数。

[1]https://dumps.wikimedia.org/
[2]https://github.com/github/CodeSearchNet

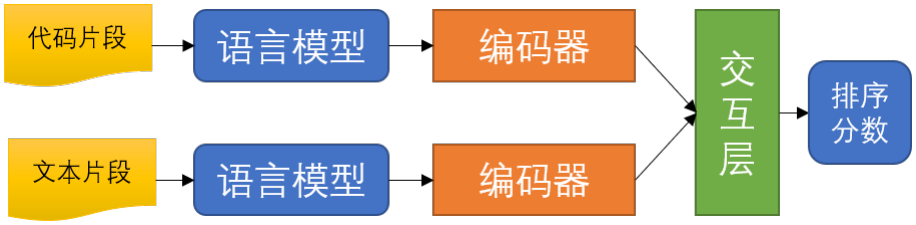

Figure 4: 排序模型结构

为了研究标识符规范性对于模型可理解性的影响，本文使用Glove(Pennington et al., 2014)、Roberta(Liu et al., 2019)和CodeBert(Feng et al., 2020)三种不同的语言模型获取代码和文本片段的语义表示。每个语言模型会重复多次实验来确保结果的可信度。

开源项目协作评估

为确保项目对比公平性。开源项目评估部分，在编程语言方面与CodeSearchNet数据集保持一致。本文选择Golang、Java、Javascript、Php、Python、Ruby六种语言的开源项目进行分析。项目选择方面，本文通过Github平台话题API[3]进行数据采集。我们依次使用六种语言作为主题关键字搜索Github平台中的开源项目，同时按照收藏（Star）数目进行排序，分别选择综合排名前100和后200的项目作为评估目标数据。

项目排名	项目数目	平均收藏	平均分支	平均观察	平均大小
前100	93	19142	3810	19142	112445
后200	169	0	0	0	6002

Table 6: Github项目数据统计

如表6所示，在下载软件项目后我们根据不同编程语言对应扩展名对各个项目中的文件进行了筛选。剔除部分教程项目和博客项目后，分别得到六种语言下平均93、169个项目进行评估。为了避免干扰，代码内注释文本、硬编码字面值和数字都被清除。可以看到，排名前100的项目和后200的项目平均大小差异较大，而规范性指标在项目规模较小评估能力较弱。为了避免项目大小差异带来的规范性指标计算失准现象，本文使用折扣因子对规模较小项目的规范新分数进行归一化，折扣因子计算方式如下：

$$d = 1 + \frac{1}{e^{-avg\_size\_last/avg\_size\_top}} \tag{8}$$

即规模较小项目的平均大小与较大项目之间的比值，经过sigmoid函数归一化后结果。

4.2 实验结果

模型可理解性分析

表8中结果为三种语言模型表示条件下，深度代码搜索模型在多轮实验后结果验证的平均分数。表中每列表示不同的标识符归一化处理方法，此处为了进一步对比本文加入未经处理的原始数据的实验结果作为基线参考。每种表示模型条件下至少重复10次实验以确保结果的可信度。从表7中可以看出，三种语言模型在多轮重复实验中，结果总体方差较小，因此实验结果具备较好的可信度。

具体而言，在汇总表8数据时，本文首先计算同一语言模型多轮实验数据结果的组内平均值，在此基础上，将三组不同语言模型在数据集上对应的结果分数进一步平均，得到最终结果数据。特别的，在实验过程中，只对表示模型部分的语言模型进行了替换和对照，其余深度模型部分三类实验中保持了一致。因此可以认为排序模型分数直接反映了不同语言模型在当前数据上的表达能力的大小。

从表中总体来看，未经任何处理的数据集上，不同的语言模型的表示能力均受到了限制。对比未经处理的原始数据与全小写处理的方法可以看到，将标识符文本全部转为小写处理之后各个语言模型表现略有提升。

[3]https://github.com/topics

编程语言		Go	Java	Javascript	Php	Python	Ruby
全小写	Glove	4.88E-05	2.30E-04	1.27E-04	4.55E-04	1.33E-04	2.14E-04
	Roberta	2.16E-03	6.18E-03	2.66E-03	2.42E-03	2.38E-03	5.09E-03
	CodeBert	3.12E-03	4.99E-03	2.82E-03	2.40E-03	2.29E-03	4.09E-03
大写字母	Glove	9.38E-05	2.02E-04	1.32E-04	2.37E-04	9.25E-05	1.32E-04
	Roberta	2.73E-03	3.97E-03	1.27E-03	2.57E-03	2.41E-03	2.19E-03
	CodeBert	3.69E-03	5.47E-03	1.11E-03	3.17E-03	2.99E-03	2.33E-03
下划线	Glove	1.26E-05	8.80E-05	2.76E-05	2.38E-04	1.62E-04	9.94E-05
	Roberta	3.38E-03	3.50E-03	2.55E-03	2.47E-03	2.05E-03	4.51E-03
	CodeBert	2.20E-03	4.39E-03	1.95E-03	3.43E-03	1.82E-03	6.30E-03

Table 7: 三种语言模型结果方差对比

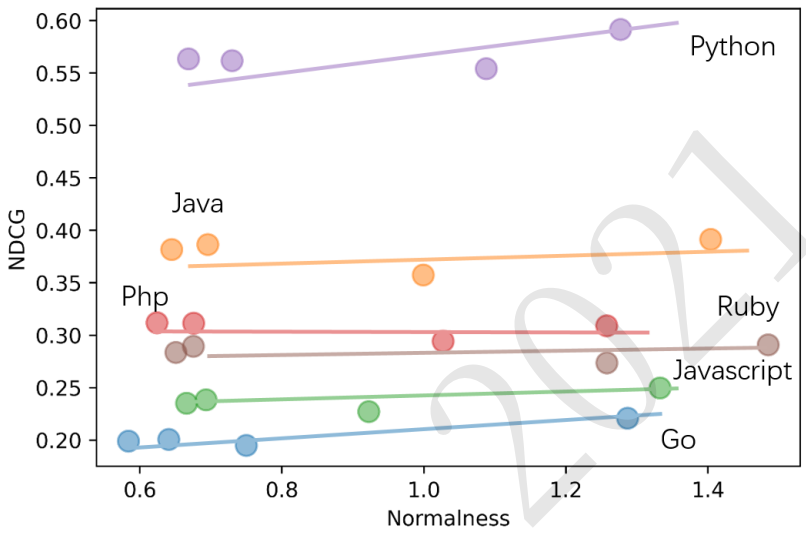

Figure 5: 规范性指标-代码搜索指标趋势图

而CodeSearchNet数据集上的规范性指标如表9中所示。通过大写字母分隔以及下划线分隔处理后，语言模型对于代码的表达能力有了更大的提高。其中Go、Java、Javascript和Php四种编程语言中，利用大写字母进行标识符分隔时效果表现最好，而Python和Ruby则在下划线分隔方法下有更好的结果表现。可以看到，这样的趋势也符合编程语言自身的命名规范。可以看到，这样的趋势与四种语言的命名规范要求相一致，同时这一趋势和代码搜索结果中的分数情况二者趋势上保持一致。说明规范性指标对于代码理解任务表现有着一定的反映能力。

为了便于进一步进行结果趋势分析，图5将表8和表9数据作为数据点进行了描绘。其中横轴为标识符规范性指标，纵轴为代码搜索评价指标，不同颜色点代表不同编程语言类型。每种编程语言对应四个点分别表示四种不同的处理方法，直线则为数据点的简单线性拟合结果。从图中可以看到，规范性指标和代码搜索模型指标之间基本呈现正相关关系，说明规范性指标对自然语言模型在代码理解任务上的适用性有着指示作用。

人可理解性分析

人可理解部分，本文对比了来自Github开源项目数据的正规性分数情况。如图6所示，左右两部分箱线图分别描述两种标识符处理方法下排名靠前的项目与靠后项目规范性指标之间的对比。可以看到，排名靠前的项目其规范性指标明显优于一般的软件项目。

综上所述，实验部分主要从模型可理解性和人可理解性两个角度对标识符规范性指标进行了研究。模型可理解性部分，本文将不同语言模型在不同标识符归一化方法处理下数据集上的效果作为监督，对比规范性指标与表示模型结果之间的相关性。实验结果表明规范性指标能够反映自然语言模型在代码搜索任务上的适用性。在结合自然语言模型研究源代码理解任务时，

编程语言	原始数据	全小写	大写字母分隔	下划线分隔
Go	0.198963	0.200572	**0.220826**	0.194871
Java	0.381667	0.386410	**0.391348**	0.357383
Javascript	0.235097	0.238334	**0.249630**	0.227120
Php	0.311828	0.311420	**0.308960**	0.294279
Python	0.563451	0.561743	0.554116	**0.591499**
Ruby	0.283529	0.289381	0.273584	**0.290804**

Table 8: 六种语言NDCG指标结果

编程语言	原始数据	全小写	大写字母分隔	下划线分隔
Go	0.584051	0.640788	**1.286950**	0.750009
Java	0.644938	0.695787	**1.403962**	0.999190
Javascript	0.665856	0.693344	**1.332600**	0.922448
Php	0.624202	0.675876	**1.257561**	1.027156
Python	0.668583	0.729964	1.087938	**1.277058**
Ruby	0.650807	0.675485	1.257791	**1.484946**

Table 9: CodeSearchNet数据集规范性指标

可以使用规范性指标作为参考，以减少实验过程中探索尝试次数。

人可理解部分，本文采集了不同水平的开源项目，以开源项目的受欢迎程度和协作深入程度作为监督，对比规范性指标与开源项目协作性之间的关系。实验结果表明，协作程度高、更受欢迎的项目，其规范性指标也表现越高。可以看到，规范性分数能够反映软件项目对特定编程语言中命名规范的遵守情况，因此能够对软件项目的可协作性进行评估。

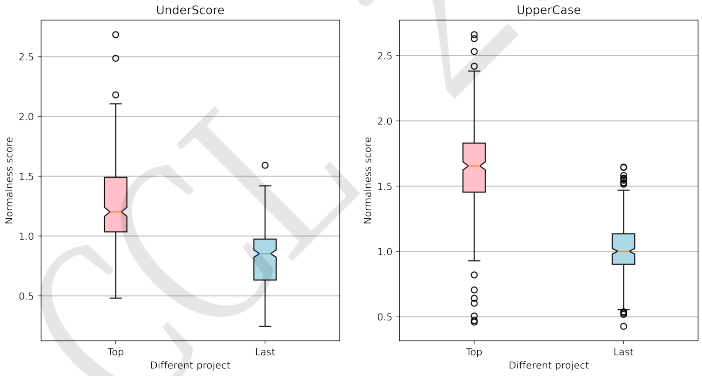

Figure 6: Github项目分数评估

5 总结和展望

总结来看，本文首先从标识符命名规范角度切入，总结开发实践中如何构造表意清晰的标识符。之后，本文从构词角度对软件标识符构造过程进行了形式化描述并引出软件标识符规范性研究。最后，本文提出一种基于自然语料库的软件标识符规范性评估方法。在开源项目和软件源代码理解相关任务上评估结果表明本文提出的规范性评价方法具有较强合理性。

可以看到，本文只在代码搜索任务上对规范性指标进行了评价，未来研究中可以继续深入在其他多项任务上进行规范性指标的验证。于此同时，开源项目的评估实验中可以看到，规范性指标对项目规模十分敏感，其原因主要在于自然语言词语的长尾分布效应。未来研究中可以尝试结合齐普夫定律对指标进行修正，以更加准确反映标识符词语的自然语言规范性情况。

参考文献

K. K. Aggarwal, Y. Singh, and J. K. Chhabra. 2002. An integrated measure of software maintainability. In *Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No.02CH37318)*, pages 235–241.

V. Arnaoudova, L. M. Eshkevari, M. D. Penta, R. Oliveto, G. Antoniol, and Y. Guéhéneuc. 2014. Repent: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering*, 40(5):502–532.

Dave W. Binkley, Marcia Davis, Dawn J. Lawrie, Jonathan I. Maletic, Christopher Morrell, and Bonita Sharif. 2013a. The impact of identifier style on effort and comprehension. *Empir. Softw. Eng.*, 18(2):219–276.

Dave W. Binkley, Marcia Davis, Dawn J. Lawrie, Jonathan I. Maletic, Christopher Morrell, and Bonita Sharif. 2013b. The impact of identifier style on effort and comprehension. *Empir. Softw. Eng.*, 18(2):219–276.

Jürgen Börstler, Michael E. Caspersen, and Marie Nordström. 2016. Beauty and the beast: on the readability of object-oriented example programs. *Softw. Qual. J.*, 24(2):231–246.

Raymond P. L. Buse and Westley Weimer. 2010a. Learning a metric for code readability. *IEEE Trans. Software Eng.*, 36(4):546–558.

Raymond P. L. Buse and Westley Weimer. 2010b. Learning a metric for code readability. *IEEE Trans. Software Eng.*, 36(4):546–558.

Raymond P.L. Buse and Westley R. Weimer. 2010c. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558.

Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2010a. Exploring the influence of identifier names on code quality: An empirical study. In Rafael Capilla, Rudolf Ferenc, and Juan C. Dueñas, editors, *14th European Conference on Software Maintenance and Reengineering, CSMR 2010, 15-18 March 2010, Madrid, Spain*, pages 156–165. IEEE Computer Society.

Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2010b. Exploring the influence of identifier names on code quality: An empirical study. In *2010 14th European Conference on Software Maintenance and Reengineering*, pages 156–165.

Breck Carter. 1982. On choosing identifiers. *SIGPLAN Not.*, 17(5):54–59, May.

Shouki A. Ebad and Danish Manzoor. 2016. An empirical comparison of java and c# programs in following naming conventions. *Int. J. People-Oriented Program.*, 5(1):39–60, January.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings, EMNLP 2020, Online Event, 16-20 November 2020*, pages 1536–1547. Association for Computational Linguistics.

Frederick. 1995. *The mythical man-month - essays on software engineering (2. ed.)*. Addison-Wesley.

Golang. 2021. Effective go - the go programming language. https://golang.org/doc/effective_go, 04. (Accessed on 04/16/2021).

Google. Google-style-guide. https://google.github.io/styleguide/. (Accessed on 03/28/2021).

Google. 2021a. Google c++ style guide. https://google.github.io/styleguide/cppguide.html#Naming, 04. (Accessed on 04/16/2021).

Google. 2021b. Google java style guide. https://google.github.io/styleguide/javaguide.html#s5-naming, 04. (Accessed on 04/16/2021).

Google. 2021c. Google javascript style guide. https://google.github.io/styleguide/jsguide.html#naming, 04. (Accessed on 04/16/2021).

Google. 2021d. styleguide — style guides for google-originated open-source projects. https://google.github.io/styleguide/pyguide.html#316-naming, 04. (Accessed on 04/16/2021).

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 933–944. ACM.

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 837–847. IEEE Press.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436.

Y. Jiang, H. Liu, J. Zhu, and L. Zhang. 2020. Automatic and accurate expansion of abbreviations in parameters. *IEEE Transactions on Software Engineering*, 46(7):732–747.

Dawn J. Lawrie and Dave W. Binkley. 2018. On the value of bug reports for retrieval-based bug localization. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*, pages 524–528. IEEE Computer Society.

Dawn Lawrie, Henry Feild, and David Binkley. 2007a. An empirical study of rules for well-formed identifiers: Research articles. *J. Softw. Maint. Evol.*, 19(4):205–229, July.

Dawn J. Lawrie, Henry Feild, and David W. Binkley. 2007b. An empirical study of rules for well-formed identifiers. *J. Softw. Maintenance Res. Pract.*, 19(4):205–229.

Xiaochen Li, He Jiang, Dong Liu, Zhilei Ren, and Ge Li. 2018. Unsupervised deep bug report summarization. In Foutse Khomh, Chanchal K. Roy, and Janet Siegmund, editors, *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, pages 144–155. ACM.

Ben Liblit, Andrew Begel, and Eve Sweetser. 2006. Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the 18th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2006, Brighton, UK, September 7-8, 2006*, page 11. Psychology of Programming Interest Group.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692.

T.J. McCabe. 1976. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.

Microsoft. 2021. Naming guidelines - framework design guidelines — microsoft docs. https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/naming-guidelines, 04. (Accessed on 04/16/2021).

pear. 2021. Manual :: Naming conventions. https://pear.php.net/manual/en/standards.naming.php, 04. (Accessed on 04/16/2021).

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1532–1543. ACL.

P. Pirapuraj and I. Perera. 2017. Analyzing source code identifiers for code reuse using nlp techniques and wordnet. In *2017 Moratuwa Engineering Research Conference (MERCon)*, pages 105–110.

Sarah Rastkar, Gail C. Murphy, and Gabriel Murray. 2014. Automatic summarization of bug reports. *IEEE Trans. Software Eng.*, 40(4):366–380.

rubocop. 2021. rubocop/ruby-style-guide: A community-driven ruby coding style guide. https://github.com/rubocop/ruby-style-guide#naming-conventions, 04. (Accessed on 04/16/2021).

C. Y. Suen. 1979. n-gram statistics for natural language understanding and text processing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):164–172.

Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 269–280, New York, NY, USA. Association for Computing Machinery.

Yanqing Wang, Chong Wang, Minghui Chen, Sijing Yun, and Minjing Song. 2014. How are identifiers named in open source software? about popularity and consistency. *CoRR*, abs/1401.5300.

Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 87–98. ACM.

Wikipedia contributors. 2021. Naming convention (programming) — Wikipedia, the free encyclopedia. [Online; accessed 29-March-2021].

Yuxiang Zhu and Minxue Pan. 2019. Automatic code summarization: A systematic literature review. *CoRR*, abs/1909.04352.

张静宣;江贺;. 2020. 代码标识符归一化研究现状及发展趋势. 计算机科学, 47(03):1–4.