# 基于方法约束关系的代码预测模型

方文渊 刘 琰 朱 玛
(数学工程与先进计算国家重点实验室 郑州 450000)

**摘 要** 最新的研究表明,从大量源代码中提取代码特征,建立统计语言模型,对代码有着良好的预测能力。然而,现有的统计语言模型在建模时,往往采用代码中的文本信息作为特征词,对代码的语法结构信息利用不充分,预测准确率仍有提升空间。为提高代码预测性能,提出了方法的约束关系这一概念;在此基础上,研究Java对象的方法调用序列,抽象代码特征,构建统计语言模型来完成代码预测,并研究基于方法约束关系的代码预测模型在Java语言中的适用范围。实验表明,该方法较现有的模型提高了8%的准确率。

**关键词** 统计语言模型,方法的约束关系,代码预测,方法调用
**中图法分类号** TP311.5 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2019.01.034

## Code-predicting Model Based on Method Constraints

FANG Wen-yuan LIU Yan ZHU Ma
(State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450000, China)

**Abstract** The state-of-the-art study shows that extracting the code features from a large amount of source codes and building the statistical language model have good predictive ability for the codes. However, the present methods still can be polished in the predicting accuracy, because when they build the existing statistical language model, the text information in the codes is often used as feature words, which means that the syntax structure information of the codes can not be fully utilized. In order to improve the predicting performance of the code, this paper proposed the concept of the constraint relation of methods. Based on this, this paper studied the method invocation sequence of Java objects, abstracted code features, and built the statistical language model to complete the code prediction. Moreover, this paper studied the application scope of the prediction model based on the method constraint relationship in Java language. Experiments show that this method improves the accuracy by 8% compared with the existing model.

**Keywords** Statistical language model, Method constraints, Code prediction, Method invocation

## 1 引言

在软件开发过程中,开发人员需要调用大量的API,以实现软件的各种功能,但开发人员通常无法掌握所有API的使用方法。当需要使用一个陌生的类时,即使是有经验的开发人员,也需要花费大量的时间来学习该类及其包含的数十种API的使用方法[1-3]。为提高软件开发的效率,构建代码预测模型并实现代码推荐已成为目前代码分析领域的研究热点。

已有研究表明,编程语言具有良好的重复性。Gabel等[4]观察到代码重复,并以6~40个代码元素的粒度报告语法冗余,如循环语句for(int $i=0;i<n;i++$)在许多源代码中频繁出现。Hindle等[5]的研究表明,这种代码规律可以通过自然语言处理技术(例如N-gram方法)进行捕获,并依此建立模型,实现代码预测与推荐。Raychev等[6]认为代码预测的核心是方法的预测,并提取源代码中的方法调用构建统计语言模型,取得了良好的预测结果。

相比于自然语言,编程语言在语法结构和数据类型上有着更为严格的要求,不同的对象、方法,甚至不同的参数之间,都可能存在着一定的约束关系,这些约束关系体现了语言本身的规则要求和开发人员潜在的思维逻辑。充分利用这些约束关系可以使统计语言模型突破单纯依靠自然语言中句子出现概率的局限性,使代码预测更符合开发人员的编程思维。但在现阶段的研究中,基于自然语言技术构建的统计模型往往缺少对这些约束关系的研究与利用,使得预测的准确率仍有提升的空间。

针对上述问题,本文将进一步增加对约束关系的研究和利用,并基于方法约束关系构建代码预测模型,研究通过自然语言处理技术预测面向对象编程语言中的方法调用。此外,本文还对该类模型的适用范围做了初步探索。

本文的主要贡献有:1)创新性地提出方法的约束关系概念,并将其应用到源代码统计语言模型中;2)提出基于方法约束关系的代码预测模型CPMMC(Code-Predicting Model based on Method Constraints);3)对基于约束关系的模型的适用性进行初步探索。

到稿日期:2017-12-17 返修日期:2018-03-29 本文受国家重点研发计划基金(2017YFB0802900)资助。
**方文渊**(1989-),男,硕士生,主要研究方向为大数据分析;**刘 琰**(1979-),女,副教授,硕士生导师,主要研究方向为大数据分析、网络数据分析,E-mail:20403686@qq.com(通信作者);**朱 玛**(1981-),女,讲师,主要研究方向为计算机网络。

本文第 2 节介绍代码推荐领域的研究现状；第 3 节提出方法的约束关系概念，并描述本文模型需要解决的问题；第 4 节介绍 CPMMC 模型；第 5 节介绍实验方法与结果；最后总结全文。

## 2 研究现状

早期的研究人员尝试通过规则匹配、代码搜索等方法来完成代码预测，但是一直难以得到较高的准确率。随着计算机性能的提高以及自然语言处理技术的日益成熟，基于代码结构或者自然语言处理来构建代码预测模型以实现代码预测逐渐成为研究热点。

Holmes 等[7]提出的代码推荐系统将开发中的代码结构与示例中的代码相匹配，由当前编写代码的前缀组成隐式查询，在样本库(例如现有项目)上执行搜索，但没有考虑时间信息(如方法调用的顺序等)。Zhong 等[8]提出了 API 挖掘框架 MAPO，使用静态分析来提取常见的 API 使用模式。MAPO 采用简单的静态分析，然后用一种查找常用 API 序列的算法为用户推荐代码片段。Han 等[9]基于隐马尔可夫模型(HMM)从用户提供的缩写中推断下一个代码元素，并发现超过 98% 的代码可以从首字母缩写中解析出来。Mishne 等[10]基于 typestate 提出了一种可用于代码推荐的代码搜索技术来向用户推荐方法名称。

Allamanis 等[11]通过抽象语法树构建代码预测模型。Nguyen 等[12]提出一种基于图形的统计语言模型 GraLan，该模型可以从源代码语料库中学习，并通过计算给定子图形的外观概率来完成代码推荐。Hsiao 等[13]基于程序依赖图构建模型，计算 JavaScript 程序的概率模型。

Hindle 等[5]于 2012 年概述了一个愿景，即希望自然语言处理技术能在各种编程任务中发挥重要作用，如代码推荐、代码搜索、属性计算等；同时提出了一个基于 Eclipse 构建的简单代码推荐方案，它使用 N-gram 模型来预测下一个代码元素；另外，还对该方案的优点和缺点进行了分析。Nguyen 等[14]将源代码中的所有元素抽象为 13 个角色，并结合代码主题构建了自然语言模型。Raychev 等[6]开创性地将一个对象涉及到的方法调用序列作为研究单元，抽取对象名、方法名、对象和方法的关系等 3 个特征词建立统计语言模型，并通过寻找频率出现最高的 $N$ 元组来预测该对象接下来的方法调用，取得了良好的效果。

由于大规模源代码带来的复杂性，基于规则匹配、代码搜索的方法难以捕获一些潜在的规则，预测的准确率普遍不高；基于程序结构的方法在预测准确率上取得了重大突破，但是复杂的模型必然带来昂贵的时空代价。例如，Mishne 等[10]提出的方法需要 3 h 才能完成 1% 的训练数据，而 Raychev 等[6]的模型只需要不到 1 min 的时间。基于自然语言处理的方法虽然在时空复杂度方面有着较大优势，但是由于没有充分利用源代码中的语法结构信息，使得准确率仍有提升的空间。其中 Raychev 等[6]的研究成果考虑了一部分对象与方法之间的约束关系，取得了非常好的效果；但是其缺少对这种约束关系的系统分析，并且未考虑编程个性化对模型的影响，需要进一步的优化。

## 3 问题描述与相关定义

自然语言处理技术已被证实可以成功运用到代码预测中。相比于自然语言，编程语言中更为严格的语法结构和数据类型约束为代码预测提供了有利的条件。本文希望在预测面向对象编程语言中的方法调用时更好地利用这些约束，但是如何构建模型来描述这些约束信息并对其进行利用仍是一个棘手的问题。本节将提出方法的约束关系这一概念并分析构建模型时需要解决的问题。考虑到不同编程语言之间的巨大差异，本文将研究对象限定为 Java 语言。

自然语言处理技术能够在一定程度上捕获编程语言中的方法调用规律的原因是，开发人员在使用这些方法时遵循着某种思维方式。通常，程序中的代码不是独立存在的，其前后的若干行代码共同决定了该代码的出现和使用方式；另一方面，两行代码的距离越近，相互之间的影响往往就越大。基于这些分析，本文提出了面向对象编程语言中方法的约束关系这一概念，并给出了定义。

**定义 1**(方法的约束关系) 在面向对象的编程语言 $l$ 中，对于任意方法 $m_1$ 和 $m_2$，如果 $m_2$ 的使用方式受到 $m_1$ 的影响，就称 $m_1$ 和 $m_2$ 之间存在约束关系。

方法的约束关系可以分成两类：方法的内部约束关系和方法的外部约束关系。

**定义 2**(方法的内部约束关系) 在面向对象的编程语言 $l$ 中，对于任意方法 $m_1$ 和 $m_2$，如果 $m_2$ 的使用方式受到 $m_1$ 的影响，且存在一个对象 $o$ 满足 $m_1, m_2 \in o$，那么称 $m_1$ 和 $m_2$ 之间存在内部约束关系。

**定义 3**(方法的外部约束关系) 在面向对象的编程语言 $l$ 中，对于任意方法 $m_1$ 和 $m_2$，如果 $m_2$ 的使用方式受到 $m_1$ 的影响，且不存在一个对象 $o$ 满足 $m_1, m_2 \in o$，那么称 $m_1$ 和 $m_2$ 之间存在外部约束关系。

在提出概念后，本文需要基于方法的约束关系构建代码预测模型，以实现对面向对象的编程语言方法调用的预测。因为需要预测的是对象的方法调用，所以在模型中如何表示一个方法就显得尤为重要。在此，我们给出代码特征词的概念。

**定义 4**(代码特征词) 在代码模型中，为表征某一次方法调用语句 $M$ 的语法结构信息，从该语句中提取若干特征 $\{t_1, t_2, \cdots, t_n\}$ 构成的特征集合 $T$。

构建代码模型，是为了将复杂多变的程序代码转化为统一范式的代码数据，并保留所需要的代码信息，为下一步的分析提供支持。但是随着软件工程领域的不断发展，程序源代码愈发复杂和庞大，不同开发人员的编程风格也存在较大差异，这给研究人员带来了巨大挑战。在模型的构建过程中需要考虑以下问题。

1)不是所有方法之间都存在约束关系，即便存在约束关系，关系的强弱也不尽相同。首先需要确定这种约束关系的范围，更准确地说应该是尽量找到与一个方法存在强约束关系的其他方法。

2)为了能应用于大规模开源代码，应避免模型的数据规模随着代码的增加而呈现几何式增长。

3)不同的约束关系有着不同的主体以及表现形式，模型

需要基于合适的代码特征词，用一种通用的形式描述，并区分不同的约束关系。

4）软件开发是一个主观过程，开发人员在命名对象时存在个性化问题。对于两条简单的语句 int len＝str.length() 和 int l＝s.length()，虽然两者的表现形式存在差异，但是实质上它们无论是调用的规则还是实现的功能都没有任何区别，更代表着同一种编程逻辑，在模型中应该用同一种形式描述它们。

5）方法的约束关系往往出现在不同的代码语句中，也就是说，单纯地依靠方法约束关系的模型难以描述同一行代码中不同代码元素之间的关系，这就需要对代码的特征词进行拓展。

针对问题 1）和问题 2），本文从模型结构的角度进行研究，把代码视为对象和方法构成的网络，根据网络的性质，将约束关系的范围定为同一个对象涉及的方法。为了控制模型的规模，采用对象的序列作为模型的基本单元，并将代码抽象为这种序列的集合。针对问题 3）－问题 5），本文从代码特征词的角度进行研究，从约束关系的主体和表现形式中抽取合适的特征词来解决问题 3）；通过将存在个性化命名问题的对象抽象为类来解决问题 4）；通过分析同一行代码中不同代码元素之间的关系，增加返回类型作为拓展特征词来解决问题 5）。该内容将在 4.1 节中进行详细分析。

## 4　CPMMC 模型

本文提出一种基于方法约束关系的代码预测模型 CPMMC，并从两个方面对其进行介绍：1）代码的抽象，解释 CPMMC 如何将复杂的面向对象的源代码抽象为结构化的代码特征词集合；2）方法的预测，解释 CPMMC 如何基于代码特征词集合构建统计语言模型来实现方法的预测。此外，本文还将尝试研究模型的适用范围。

### 4.1　代码抽象

代码抽象是构建模型的核心，它确定了模型的整体结构以及方法调用在模型中的表现形式（代码特征词）。

#### 4.1.1　模型结构

在确定模型的整体结构之前，我们需要分析方法约束关系的范围，也就是尽可能寻找强约束关系。Alan Kay 曾对面向对象语言的特性进行分析，并提出程序是对象的集合，它们通过发送消息来告知彼此所要做的事[15]。因此，Java 程序可以被看作是一个以对象为节点、方法调用（消息传递）为边的网络。基于网络的基本特征，本文可以定性地认为，同一个节点的边之间存在较为紧密的关系。也就是说，通常情况下，同一个对象涉及的方法调用之间存在较强的约束关系。因此，本文将模型的基本结构定义为同一个对象涉及的方法调用序列。

在 CPMMC 模型中，源代码被抽象为 3 个层次，形式化表示为：

$$Code=\{Class_1,Class_2,\cdots,Class_n\} \tag{1}$$

$$Class=\{Seq_{Object1},Seq_{Object2},\cdots,Seq_{Objectn}\} \tag{2}$$

$$Seq=Method_1\cdot Method_2\cdot\cdots\cdot Method_n \tag{3}$$

其中，$Class_n$ 表示第 $n$ 个类涉及到的代码语句的集合，$Seq_{Objectn}$ 表示同一个类中第 $n$ 个对象涉及到的方法调用序列，$Method_n$ 则表示同一个方法调用序列中的第 $n$ 个方法调用。CPMMC 模型将源代码转化为若干个方法调用序列 $Seq$ 的集合，每一个序列都描述了一个对象涉及到的方法调用语句。

#### 4.1.2　基于方法约束关系的特征词选取

基于 4.1.1 节的内容，本文从约束关系的主体、表现形式以及约束关系的拓展 3 个层次分析方法调用语句 *Method*，并提取代码特征词对其进行描述，如图 1 所示。

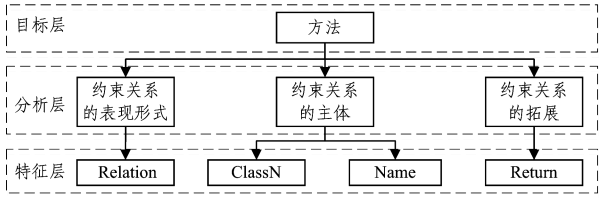


图 1　CPMMC 模型代码特征词的分析过程

Fig. 1　Code feature word analysis process of CPMMC

1）约束关系的主体。约束关系的主体即为存在约束关系的另一个方法调用。这个方法可以来自于对象本身，也可以来自于其他对象，因此需要通过该对象的对象名 *ObjectN* 来表示。考虑到参数列表（包括参数个数和类型）和方法名（它们合起来被称为“方法签名”）唯一地标识出某个方法[15]，方法名 *Name* 和参数列表 *Parameter* 也被定为候选特征词。为了解决开发人员在命名对象时存在的个性化问题，本文将对象名 *ObjectN* 抽象为类名 *ClassN*。经过实验分析，直接将参数列表加入代码特征词中并不能提高模型的准确性，因此参数列表 *Parameter* 最终也被排除。经过上述分析，代表约束关系主体的特征被确定为〈*ClassN*，*Name*〉。

2）约束关系的表现形式。约束关系的表现形式在代码中体现为方法与 $Seq_{Objectn}$ 中的对象 *Objectn* 的关系，这种关系在 Java 语言中分为 3 种情况：对象为方法的调用主体、对象为方法的参数、将方法的返回值赋值给对象。为了描述这 3 种情况，本文借鉴了 Raychev 等[6]的研究成果，用一个占位符 *Relation* 来表示。

3）约束关系的拓展。同一行代码中不同代码元素之间也存在着相互影响的关系，其中赋值符号所代表的关系最强。为了体现这种关系，本文拓展了方法的返回值 *Return* 作为方法约束关系的补充。

综上所述，一次方法的调用可以通过一个四元组来表示，形式化表示为：

$$Method=\langle ClassN,Name,Return,Relation\rangle \tag{4}$$

其中，*Method* 为对象的一个方法调用，*ClassN* 为类名，*Name* 为方法名，*Return* 为方法的返回类型，*Relation* 为对象和方法的关系。取值方式如下：

$$Relation=\begin{cases}0, & \text{对象为方法的调用主体}\\ n(n\in N), & \text{对象为方法的第 } n \text{ 个参数}\\ ret, & \text{方法的返回值赋值给对象}\end{cases} \tag{5}$$

此外，以下两种常见的情况会对抽取序列 $Seq_{Objectn}$ 产生影响：1）当出现条件选择语句（如 if/else，switch/case）时，因条件的不同，程序执行的是若干条完整且相互独立的指令，因此需要根据条件的不同构建若干条相互独立的代码序列；

2)当同一个对象出现在不同的代码结构体中时,为了降低复杂度,适当的结构体限制变得必不可少。考虑到程序中的代码与其前若干行代码紧密相关[5],而且耦合最紧密的代码语句往往在同一函数内,本文将代码序列的范围定为函数体,同一对象出现在多个函数体的情况被视为多个独立的代码序列。

Stack overflow[16]网站中的一段代码片段如下:

```
SmsManager smsMgr=SmsManager.getDefault();
int length=message.length();
if (length > MAX_SMS_MESSAGE_LENGTH)
{
  ArrayList<String> msgList=smsMgr.divideMsg(message);
  smsMgr.sendMultipartTextMessage(msgList);
}
Else
{
  smsMgr.sendTextMessage(message);
}
```

该片段可被抽象为 3 个对象的代码特征词序列,如表 1 所列。

表 1 CPMMC 模型的代码特征词序列

Table 1 Code feature word sequence of CPMMC

| 类 | 代码特征词序列 |
|---|---|
| SmsManager | {(SmsManager,getDefault,SmsManager,ret)·(SmsManager,divideMsg,ArrayList<String>,0)·(SmsManager,sendMultipartTextMessage,void,0)} |
| | {(SmsManager,getDefault,SmsManager,ret)·(SmsManager,sendTextMessage,Void,0)} |
| String | {(String,length,Int,0)·(String,divideMsg,ArrayList<String>,1)} |
| | {(String,length,Int,0)·(String,sendTextMessage,Void,1)} |
| ArrayList<String> | {(ArrayList<String>,divideMsg,ArrayList<String>,ret)·(ArrayList<String>,sendMultipartTextMessage,Void,1)} |

### 4.2 方法预测

CPMMC 基于代码抽象后的代码特征词序列集合构建 N-gram 统计语言模型来实现方法的预测。统计语言模型通过将语言单位(如单词、短语、句子和文档)的发生概率赋予自然语言来捕捉自然语言的规律[17]。由于语言单位被表示为一个或多个基本符号的序列,因此通过计算这种序列的概率来构建语言模型,在形式上如定义 5 所示。

**定义 5**(语言模型) 语言模型 $L$ 是由基本单元词汇 $V$、生成过程 $G$ 和似然函数 $P(\cdot|L)$组成的统计模型。$P(s|L)$是 $L$ 由过程 $G$ 之后生成 $V$ 中元素序列 $s$ 的概率。

在一个确定的语境中构建语言模型时,可以使用 $P(s)$来表示 $P(s|L)$,称其为序列 $s$ 的生成概率。因此,可以简单地将语言模型视为具有每个可能序列的概率分布。

N-gram 模型是一个具有两个假设的统计语言模型:1)一个序列可以从左到右生成;2)该序列中的单词的产生概率仅取决于其本地语境。$n$ 个单词的序列称为 N-gram,当 $n$ 的值为 1,2 或 3 时,该模型被称为 unigram,bigram 或 trigram。

现有研究已经证明,基于源代码的 N-gram 模型是合理的[5],即下一个代码特征词是可以根据先前的代码特征词进行预测的。例如,在源代码中,FileWriter 类的实例化对象 fw 分别调用了以下方法:

```
fw=new FileWriter(filepath);
fw.write(str);
```

那么,这组代码序列可以被视为下一句代码的上下文,并用于预测接下来对象 fw 调用的方法,如 fw.close()。

假设从左到右产生代码特征词,序列 $s=m_1\cdot m_2\cdot m_3\cdot\cdots\cdot m_{i-1}$之后的下一个代码特征词 $c=m_i$ 的概率可被表示为:

$$P(c|s)=P(m_i|m_1\cdot m_2\cdot\cdots\cdot m_{i-1}) \tag{6}$$

根据马尔科夫假设,条件概率 $P(c|s)$可被近似计算为:

$$P(c|s)=P(m_i|m_{i-n+1}\cdot m_{i-n+2}\cdot\cdots\cdot m_{i-1}) \tag{7}$$

其中,$m_{i-n+1}\cdot m_{i-n+2}\cdot\cdots\cdot m_{i-1}$是由序列 $s$ 的最后 $n-1$ 个代码特征词构成的子序列。通过这种近似,模型只需要计算和存储涉及最多 $n$ 个连续代码特征词的条件概率。

CPMMC 采用 trigram 模型,将一个长度为 $l$ 的代码特征词序列 $Seq_{Objectn}$ 切分为 $l-2$ 个连续的序列。

$$Seq_{Objectn}\mapsto\{Seq_1^t,Seq_2^t,\cdots,Seq_{l-2}^t\} \tag{8}$$

$Seq_i^t$ 表示由 3 个连续的代码特征词构成的序列:

$$Seq_i^t=Method_1^i\cdot Method_2^i\cdot Method_3^i \tag{9}$$

在进行切分后,源代码 $Code$ 成为由 $n$ 个 $Seq^t$ 构成的集合:

$$Code\mapsto\{Seq_1^t,Seq_2^t,\cdots,Seq_n^t\} \tag{10}$$

在此基础上,代码特征词序列 $Method_1^i\cdot Method_2^i$之后生成 $Method_3^i$的概率为:

$$P(Method_3^i)\approx\frac{count(Method_1^i\cdot Method_2^i\cdot Method_3^i)+\alpha}{count(Method_1^i\cdot Method_2^i)+V\cdot\alpha} \tag{11}$$

其中,$\alpha$ 和 $V$ 为平滑参数,以处理极小值甚至 0 值出现的情况。

因为存在数据稀疏的问题,对于少数无法构建 trigram 模型的情况,CPMMC 会采用 bigram 甚至 unigram 模型。

### 4.3 模型的合理性分析

CPMMC 模型的基本思想是把编程代码视为自然语言,将对象视为主语,对象涉及到的方法调用视为谓语,然后构建统计语言模型来预测对象的行为。该思想已经在 Hindle[5]、Nguyen 等[14]、Raychev 等[6]的研究中得到了证实。在此基础上,CPMMC 系统分析了方法的约束关系,提取代码特征词,构建模型。本文认为,方法之间的约束关系对方法的使用方式有着非常重要的作用,是一个值得深入研究的方向。

例如,Java 程序员为了实现文件读取操作,编写了一段不完整的代码,具体如下:

```
BufferedReader reader=
    new BufferedReader(new FileReader(file));
String tempString=reader
reader.x
```

reader 对象的下一个方法 $x$ 的使用受之前两个方法(初始化、readLine)的约束。初始化、readLine 和 $x$ 相互之间一定符合某种逻辑,并且往往会在不同代码中多次出现。但是,单纯依靠方法的约束关系很难判断出 reader 对象是否还会继续读取文件。在加入拓展特征词后,CPMMC 可以通过 $x$ 的

返回类型 void 排除继续使用 readLine 方法的可能性。经过统计模型,可以得到 $x$ 为 close 方法的概率较高。

经过上述分析,CPMMC 预测方法具有合理性,本文将会在 5.4 节通过实验进一步论证。

### 4.4　模型的复杂度分析

CPMMC 在训练过程中主要分为代码抽象和 trigram 模型构建。

对于一个大小为 $N$ 的训练集,其包含的对象个数为 $o$,对象涉及到的方法数为 $m$。

在代码抽象过程中,假设提取方法的特征词所需要的时间为 $t$,时间复杂度 $T_1$ 可表示为:

$$T_1=\sum_{i=0}^{o}(m_i\times t_i) \tag{12}$$

根据编程语言的一般规律,$o$ 与 $N$ 正相关,存在线性关系,式(12)可近似表示为:

$$T_1\approx o\times\bar{m}\times\bar{t}\approx N\times C \tag{13}$$

其中,$\bar{m}$为平均每个对象涉及到的方法数,$\bar{t}$ 为每个方法提取特征词所需要的平均时间。因此,代码抽象过程的时间复杂度为:

$$T_1(N)=O(N) \tag{14}$$

在构建 trigram 模型的过程中,假设构建并存储一个新的 trigram 需要的时间为 $\tau$,新 trigram 出现的概率为 $p$,那么构建模型的时间复杂度 $T_2$ 可表示为:

$$T_2=\sum_{i=0}^{o}((m_i-2)\times\tau_i\times p_i) \tag{15}$$

式(15)可以近似表示为:

$$T_2\approx(\bar{m}-2)\times\bar{\tau}\times\sum_{i=0}^{o}(p_i)\approx C\times\sum_{i=0}^{o}p_i \tag{16}$$

因为代码中的方法序列存在一定的重复性,所以 $p_i$ 与 $i$ 的大小负相关,当 $i$ 无限大时,$p_i$ 趋于 0,即 $p$ 与 $N$ 存在负相关。显然,时间复杂度 $T_2$ 小于 $N\times C$,即:

$$T_2(N)<O(N) \tag{17}$$

### 4.5　模型的适用范围

本文基于方法的约束关系提出 CPMMC 模型。事实上,方法约束关系的强弱是存在差异的,例如 open/close 往往成对出现。换言之,在同一个对象中,方法 open 和 close 之间存在巨大的约束关系。因此,研究方法约束关系的强弱对这类模型的影响,并进一步讨论这类模型在代码预测领域的适用范围,是非常必要的。

根据功能的不同,Java 语言中的类可以分成 3 类,分别为数据结构类、内部功能类和外部接口类,如表 2 所列。

表 2　Java 类的分类

Table 2　Classification of Java class

| 类型 | 描述 | 代表类 |
|---|---|---|
| 数据结构类 | 定义常见类型的类 | java.lang.Integer<br>java.util.List |
| 内部功能类 | 实现编程语言本身功能的类 | java.io.StringWriter<br>java.lang.Thread |
| 外部接口类 | 调用或访问外部程序的类 | java.io.FileInputStream<br>java.sql.DriverManager |

一般情况下,外部接口类因为要与外部程序(如文件系统、数据库、浏览器等)进行交互,需要遵循一些严格的规范(如文件的打开和关闭),经常需要连续调用若干方法来实现某种功能,方法之间的约束关系较强;数据结构类中的方法往往实现一些最简单的数据操作,相互的依赖性不强,调用灵活,方法之间的约束关系较弱;而内部功能类虽然不需要遵循外部程序提供的接口,但是相比数据结构类的方法,调用过程较为复杂,方法约束关系的强弱通常在另外两类之间。

本文将在 5.6 节通过实验对这 3 种类型的 Java 类进行分析研究。

## 5　实验与结果分析

根据 CPMMC 模型,以开源的 Java 代码为样本,对代码中的方法调用进行预测,以验证 CPMMC 模型的效果和性能,分析 CPMMC 模型的适用范围。

### 5.1　数据集合

验证模型在代码预测中的效果需要大量的面向对象的源代码。本文实验在 Github[18] 网站上获取了评分在 1000Stars 以上的 11 个 Java 开源代码,代码总数为 139KLOC。为了去除开发人员自定义的类对预测的干扰,实验采集了 JavaTM Platform Standard Ed.7 版本中定义的 4024 个类,以及这些类包含的 51641 个方法,并将其设定为实验预测的范围。

考虑到 Java 语言中的一些结构会对预测产生负面影响,本文基于 Java 分析工具 Soot[19] 构建程序分析框架对源代码进行预处理,主要分为以下 3 种情况:1)针对别名现象,采用 Spark 框架对程序进行别名分析,将指向同一地址的对象标记为相同对象;2)针对条件选择结构,根据条件的不同将函数体分成若干个相互独立的、不含有条件选择结构的函数体;3)针对循环结构,将循环的迭代次数统一限定为 2。

### 5.2　实验设计

实验将 90%的源代码作为训练集,剩下的 10%作为测试集。在测试集中随机删除 100 个方法调用,将其标记为待预测的点 P.P(Predicting Point),如表 3 所列,并确保调用 P.P 的对象在之前至少涉及到 2 个方法的调用。

表 3　测试集中删除方法的示例

Table 3　Example of delete method in test set

| | |
|---|---|
| 原代码片段 | FileOutputStream in=new FileOutputStream(file);<br>try {<br>　in.write(bt,0,bt.length);<br>　in.close();<br>　System.out.println("true");<br>} catch (IOException e) {<br>　e.printStackTrace();<br>} |
| 随机删除方法后的代码片段 | FileOutputStream in=new FileOutputStream(file);<br>try {<br>　in.write(bt,0,bt.length);<br>　in.P.P //删除方法 close()<br>　System.out.println("true");<br>} catch (IOException e) {<br>　e.printStackTrace();<br>} |
| 抽象后的方法序列 | FileOutputStream in=new FileOutputStream(file);<br>in.write(bt,0,bt.length);<br>in.P.P //待预测的点 |

实验整体分为模型训练部分和代码预测部分,如图 2 所

示。在模型训练部分，对训练集进行程序分析，得到中间代码，并抽取方法的调用序列，然后对得到的序列进行 N-gram 切分，构建 CPMMC 模型。在代码预测部分，对测试集中不完整的代码片段进行程序分析，并抽取带有 P. P 的方法调用序列，构建 CPMMC 模型。然后根据训练得到的特征词序列的概率分布，给出 P. P 的推荐列表。

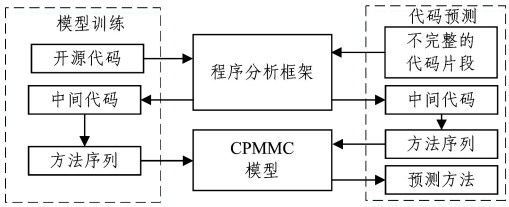


图 2 实验总体框图

Fig. 2 Overall block diagram of experiment

为了更好地评估预测的准确率，实验设计了 3 个指标：Top1，Top3 和 Top15。Top1 表示模型推荐的首个方法即为正确方法的概率，Top3 表示推荐列表中前 3 个方法包含正确方法的概率，Top15 表示推荐列表中前 15 个方法包含正确方法的概率。

### 5.3 程序分析的重要性评估

为了评估程序分析在构建模型时的重要性，本文分别构建了经过程序分析得到的 CPMMC 模型、未进行别名分析得到的 CPMMC_A 模型、未处理条件选择结构的 CPMMC_C 模型，并对 100 个相同的 P. P 进行预测。实验结果如图 3 所示。

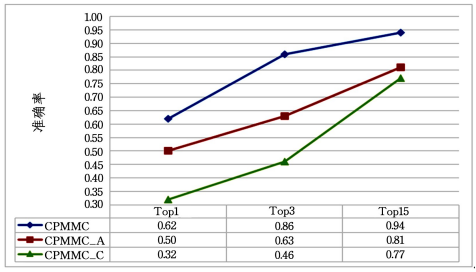


图 3 3 种模型的实验结果

Fig. 3 Experimental results of three models

通过对比实验可以看出，别名分析和对条件选择结构的处理对预测的准确率有着较大的影响，尤其是对条件选择结构的处理十分关键，这也验证了在构建模型前对源代码进行程序分析的必要性。

### 5.4 模型的先进性评估

为了评估 CPMMC 模型的先进性，将本文的模型与当前预测准确率最高的 SLANG(Statistical Language Model)模型[6]进行比较。此外，4.1 节中提到，本文没有将方法的参数列表作为代码特征词之一，为验证本文模型的合理性，我们还构建了带有参数列表的 CPMMC 模型(用 CPMMC_P 表示)。用这 3 种模型对 100 个相同的 P. P 进行预测，实验结果如图 4 所示。

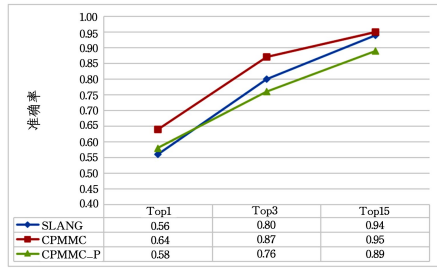


图 4 3 种模型的实验结果

Fig. 4 Experimental results of three models

可以看出，相比于 SLANG 模型，CPMMC 模型的准确率有明显的提高，Top1 从 0.56 提高到 0.64，Top3 从 0.80 提高到 0.87，Top15 从 0.94 提高到 0.95。这也验证了 CPMMC 模型在解决对象的个性化命名问题，以及加入了同一行代码中的不同元素的约束关系后，明显提高了预测的准确性。两种模型在 Top15 指标上相差不大，都达到了 94% 以上，主要是因为随着指标的放宽，这两种模型都可以有效地捕获开发人员对 Java 方法的使用规则。

通过比较 CPMMCP 模型和 CPMMC_P 模型的实验结果可以发现，CPMMC_P 模型预测的准确率有明显的下降。表面上看，加入参数列表会使模型数据更加稀疏，例如源代码中常见的 String 类，它的一个方法 indexOf 经重载后拥有 4 个同名方法，即 indexOf(int)，indexOf(string)，indexOf(int，int)，indexOf(string，int)，虽然 indexOf 可能得到较高的概率值，但是如果分散到 4 个方法中，每个方法的概率值反而不高。单从本质上分析，由于预测的目标是方法名称，并未深入到方法的重载，加入参数列表会造成过拟合现象，因此 CPMMC_P 模型的效果反而不好。

### 5.5 模型的时间复杂度评估

为了评估 CPMMC 模型的时间复杂度，分别计算了对于不同大小的训练集，代码抽象和构建 N-gram 模型所需要的时间，并与 SLANG 模型[6]进行了比较。实验结果如表 4 所列。

表 4 训练阶段所需时间

Table 4 Time required for training

| 模型 | | 训练集大小 | | |
|---|---|---|---|---|
| | | 1% | 10% | 100% |
| CPMMC | 代码抽象 | 23.737 s | 4m23 s | 43 m57 s |
| | 构建 N-gram 模型 | 1.789 s | 13.692 s | 58.051 s |
| | 总计 | 25.526 s | 4 m37 s | 44m55 s |
| SLANG | | 18.659 s | 3 m49 s | 32 m6 s |

通过实验结果可以发现，CPMMC 模型的时间复杂度随着样本量的增加呈线性增长，与 SLANG 模型相比，增加的时间在可控范围内，可以应用于大规模开源代码。

### 5.6 模型的适用范围

为了研究方法约束关系的强弱对基于方法约束关系的模型的影响，进一步讨论这类模型在代码预测领域的适用范围，本文在 4.3 节中将 Java 的类分成 3 类。对此，本文通过实验数据统计得出 Java 代码中使用频率最高的 60 个类，并根据

数据结构类、内部功能类和外部接口类分别选取 6 个有代表性的类，如表 5 所列。实验对每一个类都分别构建 CPMMC 模型并对 100 个 P. P 点进行预测，结果如图 5 所示。

表 5　3 种类型的 Java 类代表

Table 5　Representation of three types of Java classes

| 分类 | 类 |
|---|---|
| 数据结构类 | java. lang. String<br>java. lang. Integer<br>java. util. List<br>java. lang. Character<br>java. lang. Double<br>java. util. Calendar |
| 内部功能类 | java. util. Map<br>java. util. HashMap<br>java. lang. reflect. Method<br>java. util. regex. Pattern<br>java. io. StringWriter<br>java. util. Locale |
| 外部接口类 | java. io. File<br>java. io. InputStream<br>java. io. FileInputStream<br>java. net. URL<br>javax. servlet. http. HttpServletRequest<br>java. io. Reader |

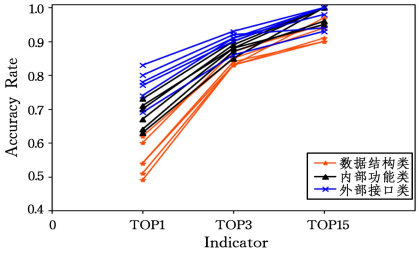


图 5　CPMMC 模型对不同 Java 类的预测结果

Fig. 5　Prediction results for different Java classes

通过实验结果可以看出，CPMMC 模型对外部接口类的预测准确率最高，而对数据结构类的预测准确率最低。考虑到一般情况下，外部接口类的方法约束关系最强，数据结构类的方法约束关系最弱，我们可以得出这样一个结论：在基于方法约束关系的代码预测模型中，预测的准确率与方法约束关系的强弱成正相关。与此同时，这也说明基于方法约束关系的代码预测模型对外部接口类的适用性最好。

**结束语**　针对现有基于自然语言的代码预测模型未能充分利用编程语言中方法之间的相互关系这一问题，本文提出了方法的约束关系这一概念，并基于此概念提出了 CPMMC 模型，以同一对象涉及的方法调用序列为研究对象，将类名、方法名、返回类型、对象和方法的关系等要素作为特征词，构建统计语言模型，对某个对象的下一个方法调用进行预测。实验结果表明，该模型能较好地利用代码中方法的约束关系，提高了代码预测的准确率，并对 Java 语言中的外部接口类有良好的预测能力。

## 参 考 文 献


[1] BECKMAN N E, KIM D, ALDRICH J. An Empirical Study of Object Protocols in the Wild[C]// European Conference on Object-Oriented Programming. Springer-Verlag, 2011: 2-26.

[2] WASYLKOWSKI A, ZELLER A. Mining Temporal Specifications from Object Usage[C]// IEEE/ACM International Conference on Automated Software Engineering. IEEE, 2009: 295-306.

[3] WEIMER W, NECULA G C. Mining temporal specifications for error detection[C]// International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, Berlin, Heidelberg, 2005: 461-476.

[4] GABEL M, SU Z. A study of the uniqueness of source code[C]// Eighteenth ACM Sigsoft International Symposium on Foundations of Software Engineering. ACM, 2010: 147-156.

[5] HINDLE A, BARR E T, SU Z, et al. On the naturalness of software[C]// International Conference on Software Engineering. IEEE, 2012: 837-847.

[6] RAYCHEV V, VECHEV M, YAHAV E. Code completion with statistical language models[C]// Acm Sigplan Symposium on Programming Language Design & Implementation. ACM, 2014: 419-428.

[7] HOLMES R, MURPHY G C. Using structural context to recommend source code examples[C]// International Conference on Software Engineering. 2005: 117-125.

[8] ZHONG H, XIE T, ZHANG L, et al. MAPO: Mining and Recommending API Usage Patterns[C]// ECOOP 2009-Object-Oriented Programming. Springer Berlin Heidelberg, 2009: 318-343.

[9] HAN S, WALLACE D R, MILLER R C. Code Completion from Abbreviated Input[C]// IEEE/ACM International Conference on Automated Software Engineering. IEEE, 2009: 332-343..

[10] MISHNE A, SHOHAM S, YAHAV E. Typestate-based semantic code search over partial programs[J]. Acm Sigplan Notices, 2012, 47(10): 997-1016.

[11] ALLAMANIS M, SUTTON C. Mining idioms from source code[C]// Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2014: 472-483.

[12] NGUYEN A T, NGUYEN T N. Graph-Based Statistical Language Model for Code[C]// IEEE/ACM, IEEE International Conference on Software Engineering. IEEE, 2015: 858-868.

[13] HSIAO C H, CAFARELLA M, NARAYANASAMY S. Using web corpus statistics for program analysis[J]. ACM Sigplan Notices, 2014, 49(10): 49-65.

[14] NGUYEN T T, NGUYEN A T, NGUYEN H A, et al. A statistical semantic language model for source code[C]// Joint Meeting on Foundations of Software Engineering. 2013: 532-542.

[15] ECKEL B. Thinking in Java (4th Edition)[M]. Prentice Hall PTR, 2005.

[16] Stackoverflow[OL]. http://www.stackoverflow.com.

[17] ROLAND H. Foundations of statistical natural language processing[M]. The MIT Press, 1999.

[18] Github[OL]. https://github.com.

[19] GAGNON V S P L E, VALLÉE-RAI R, HENDREN L. Soo—a java optimization framework[C]// Proceedings of CASCON. 1999: 125-135.