Dem代码摘要模型学到了什么

YU WANG,中国南京大学KE
WANG, 美国 Visa Research

arXiv:2303.02333v1 [cs.PL] 4 Mar 2023

研究模型所学到的模式，长期以来一直是模式识别研究的重点。阐明从训练数据中发现了哪些模式，以及这些模式如何泛化到未见数据，对于理解和推进模式识别方法至关重要。遗憾的是，绝大多数应用领域处理的都是连续数据（即 具有统计性质），从中提取的模式无法被形式化定义。例如，在图像分类中，不存在猫或狗这类标签的原则性定义。即使在自然语言中，一个词的含义也会因其所处的上下文而变化。与上述数据格式不同，程序是一种独特的数据结构，具有定义明确的语法和语义，这为形式化描述模型从源代码中学到的内容提供了绝佳机会。本文首次对代码摘要模型发现的模式进行了形式化定义（即根据方法主体预测方法名称的模型），并给出一种可靠算法，用于推导出能够形式化描述所学模式的上下文无关文法（CFG）。

我们在 PATIC 中实现了这一方法。PATIC 会生成上下文无关文法，用于概括代码摘要模型发现的模式。具体而言，我们选取了两个具有代表性的模型——code2vec 和 code2seq——来评估 PATIC。PATIC 表明，各模型提取的模式主要局限于局部的语法代码结构，几乎不包含语义含义。基于这些发现，我们展示了模式形式化定义的两种示例用途：一种用于评估鲁棒性的新方法，以及一种用于提高代码摘要模型准确率的新技术。

我们的工作开辟了一个令人振奋的新方向：研究模型从源代码中学到了什么。

附加关键词与短语：上下文无关文法、模式识别、代码摘要、深度神经网络、模型泛化能力

1 引言

随着深度学习取得重大突破，公共数据集不断增加，计算能力持续提升，机器学习模型已能为包括图像分类在内的诸多问题提供最先进的解决方案 [Krizhevsky 等，2012；Touvron 等2019], 机器翻译 [Conneau 和 Lample，2019；Devlin 等 2019], 以及游戏对弈 [Silver 等，2016、2017；Vinyals 等 2019]。

基于学习的方法之所以取得成功，很大程度上归功于其从大量数据中发现其中所呈现模式的能力。机器学习领域中一个颇具影响力的子领域一直致力于解释和可视化模型从数据中学到的模式。此外，该领域在应对人工智能（AI）面临的一些最紧迫挑战方面发挥着主导作用，因此日益受到关注。例如，可解释性很可能会成为下一代人工智能技术的核心目标。揭示模型学到了什么，是设计此类可解释人工智能系统的关键第一步。此外，从科学角度来看，剖析复杂模型的内部运作机制和行为是必要的。因为如果不能清楚理解机器学习模型如何以及为何发挥作用，那么更优模型的开发就会沦为试错。

作者地址：Yu Wang，新型软件技术国家重点实验室，计算机科学与技术系，南京大学，中国江苏省南京市，210023，yuwang_cs@nju.edu.cn；Ke Wang，安全、密码学与区块链，Visa Research，美国加利福尼亚州帕洛阿尔托，kewang@visa.com。

2021年. 2475-1421/2021年/
1-ART1 $15.00https://doi.org/

在一些值得注意的研究工作中，Chen 等人 [2006] 提出了一种上下文敏感文法，通过复合图形模板对物体配置的广泛变化进行建模。他们的方法的一项优势在于，能够解释模型在推理时从测试数据中识别出的模式；在 [Chen 等人 2006], 所讨论的布料建模案例中，模型使用表示鞋子、手部、面部等对象的模板进行预测。Zeiler andFergus [2014]提出了一种可视化技术，用于深入了解各个特征层的功能以及卷积网络的端到端运行过程；卷积网络是深度神经网络（DNN）中常用于图像分类的一类。作为一种诊断工具，该可视化技术帮助他们找到性能优于 AlexNet [Krizhevsky 等人 2012], 当时在 ImageNet 上达到业界最先进水平的模型 [Deng 等人 2009]。相关工作的详细综述将在第6节中介绍。

尽管取得了重大进展，将模型学到的模式形式化仍然是一项极具挑战性的任务。这在很大程度上是由基于学习的方法所应用的问题领域的性质造成的。模型几乎只处理连续数据，而从中学习形式化模式十分困难，甚至是不可能的。例如，迄今为止已知的机器学习模型中，没有一个致力于学习 ImageNet 中各标签的基本定义（例如熊猫、鸵鸟、金鱼等）。即使在自然语言处理领域，学习形式化模式也是一项棘手的任务，因为单词的含义可能模棱两可。

当编程似乎已成为机器学习模型（以深度神经网络为代表）的又一个热门领域时，我们必须认识到，程序是一种本质上截然不同的数据结构。具体而言，程序具有离散的本质，其语法和语义都有明确定义。从语法上看，程序的编写方式满足（即）上下文无关文法中定义的递归规则（产生式规则）。从语义上看，程序的行为满足小步语义中定义的推理规则 [Plotkin 和 Kahn 1987]。以上这些都启发了我们的工作：模型从源代码中学到的模式可以形式化。然而，我们面临一个重要挑战：如何在一个庞大的搜索空间中高效探索各种程序属性，这些属性涵盖从语法到语义的方方面面?

我们的解决方案基于一项关键且颇出人意料的观察：我们发现，许多知名代码模型的行为都呈现出这一现象。在 Wang 和 Christodorescu [2019], 的研究基础上——该研究发现，句法上微不足道且不改变语义的代码编辑，经常会导致模型改变其预测——我们观察到了一个更加令人惊讶的现象。也就是说，当要求模型对一个程序进行预测时，几乎总能将该程序缩减为极少数语句（即 ≤ 2），而模型对这些语句所作的预测与对原始程序所作的预测相同。这一发现具有两方面的重要意义。首先，(1) 这表明一个小型的局部代码窗口就足以涵盖模型为预测整个程序的属性而寻找的模式。因此，模式定义的搜索空间比预想的小了好几个数量级。其次，(2) 对于这类往往在语义上毫无意义的简单模式，可以放心忽略语义属性的谓词，从而进一步将搜索空间限制在句法属性的谓词上。基于 (1) 和 (2)，一种自然而然的模式定义思路应运而生：根据原始程序中关键语句的句法属性来综合规则。然而，这里存在一个问题：无论周围上下文如何，仅凭关键语句是否总能保留模型针对原始程序所预测的标签? 为了解决这一问题，我们找出了关键语句确实能够保留原始预测标签的有效程序集合，并据此定义了模型学到的模式。

在技术层面，我们提出“抽象、变异、具体化和总结”（Abstract, Mutate, Concertize, and Summarize），一种用于模式形式化的新方法。首先，给定一组程序 P，它们都带有由模型 M预测的标签 L，我们从 P 中的每个程序中抽象掉不会导致 M 改变其预测的语句。我们将每个剩余的代码片段称为 种子，它捕捉了M 针对标签 L所作预测的本质。其次，鉴于机器学习模型已被证实具有泛化

能力，我们推测，与某个种子相似的程序很可能会被预测为相同的标签 L。因此，我们对每个种子进行变异，以获得更多代码片段，并将其称为 变异体。第三，我们通过向每个种子和变异体中插入语句来合成完整的程序。具体而言，我们使用程序 P 所采用语言的语法，枚举一组多样的语句和表达式。随后，我们将每个合成的程序传递给模型，以获得一个标签。最后，我们推断出一个上下文无关文法，用于描述所有使 M预测标签 L 的合成程序，并将其作为模型 M 关于标签 L所学习模式的原则性定义。

我们开发了一个名为 PATIC 的工具，作为“抽象、变异、具体化与总结”方法的实现，该工具能够自动形式化两个代码摘要模型所学习的模式：code2vec [Alon 等人，2019b] 和 code2seq [Alon 等人，2019a]。代码摘要指的是这样一项任务：模型根据方法的主体推断其名称。图1 展示了一个示例。该方法的正确预测结果是 `reverseArray`。我们之所以针对代码摘要模型，是因为它们对编程语言社区产生了巨大影响。自两年前 code2vec 发表在 编程语言原理（*POPL*）以来，它不仅获得了大量引用（即 130+），还促成了若干有趣的后续研究（例如 code2seq、序列 GNN [Fernandes 等人 2019],和 LiGER [Wang 和 Su 2020]）。

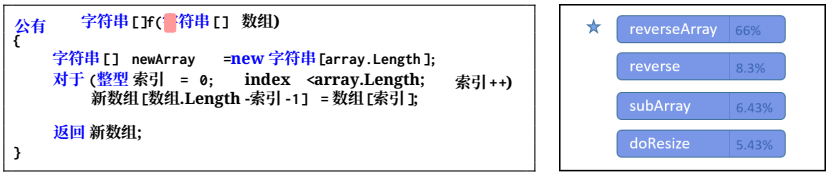


图 1。从 [Alon 等人，2019b] 中提取的一个示例方法，其名称被去除，以供模型进行预测。右侧展示了 code2vec 作出的前 4 个预测。

通过 PATIC 推断出的上下文无关文法，我们发现，两个经评估模型学习到的模式相对于 所有 Java-small、Java-med 和 Java-large 中包含的所有标签而言都很简单，这些数据集来自 [Alon 等人，2019a], 提出的三个公有、大规模、跨项目数据集，被广泛用于许多代码摘要研究。计算得出，种子中呈现的词元平均数量为 15.59。对于几乎所有方法而言，其主体中的绝大多数语句都可以被移除，而所得种子包含的语句少于两条。此外，在存在种子语句的情况下，完整程序的合成几乎不受约束；我们插入种子中的许多构造甚至与非种子语句的语义完全不相似。尽管如此，模型仍表现出强烈倾向，会保留它们为原始程序所作出的预测。我们的发现表明，code2vec 和 code2seq 都没有尝试学习给定方法的全局语义表示；相反，它们使用局部句法程序特征作为代理，以简化对该方法的记忆。

基于我们对代码摘要模型所学内容的发现，我们展示了基于上下文无关文法的模式形式化的两种示例用途。首先，我们提出一种评估代码摘要模型鲁棒性的新方法。具体而言，我们构造攻击，以揭示它们面对输入程序中微小扰动时的脆弱性。我们的思路是将改动集中在种子语句上，也就是模型主要据以进行预测的方法部分，从而影响模型对原始程序所作的预测。此外，我们还引入四种保持语义的程序变换，使我们能够为 Java-small、Java-med 和 Java-large 中每个预测正确的测试方法找到极小的扰动（即 ≤ 2 扰动程序与未扰动程序的抽象语法树之间的树编辑距离）。其次，

我们提出一种提高代码摘要模型泛化能力的新技术。秉承对抗训练的理念，[Goodfellow 等人 2015], 我们选择将为解决代码摘要模型的特定弱点而合成的程序纳入训练，以重新训练模型。具体而言，在收集模型预测错误的训练程序后，我们将这些程序中的种子注入其他程序，以合成额外的训练数据。具体来说，通过将宿主程序的标签赋给合成程序，我们引导模型将注意力转向此前未曾关注的不同句法结构，从而为模型关联到真实值铺平道路。经过这样的重新训练后，code2vec 和 code2seq 的准确度都有所提高，尤其是 code2seq，在 Java-med 和 Java-large 上取得了最先进的结果。

本文作出了以下贡献：

- 对代码摘要模型所学习的模式给出了形式化定义。
- 一种用于形式化代码摘要模型所学模式的可靠算法。假设单调性（参见定义 3.3），该算法也是完备的。
- 一种名为 PATIC 的实现，它能够自动以上下文无关文法的形式生成代码摘要模型发现的模式定义。
- 对 PATIC 在两个主要代码摘要模型 code2vec 和 code2seq 上进行了实证评估。通过 PATIC 推断出的上下文无关文法，我们发现，这两个模型所学习的模式都未能准确捕捉具有预测标签的程序的属性。
- 一种新的模型鲁棒性评估方法，能够以比以往方法少得多的尝试次数，找到扰动幅度更小的对抗样本。
- 一种提高代码摘要模型准确率的新技术，使 code2seq 在 Java-med 和 Java-large 上达到最先进的结果。

2 概述

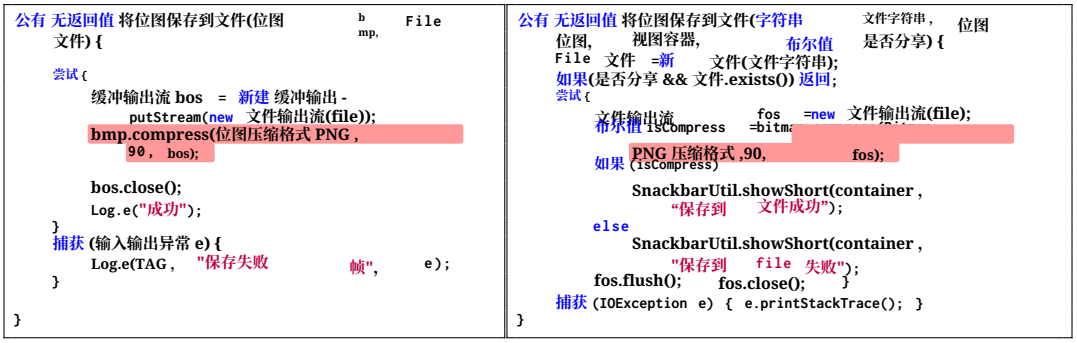
本节概述我们将代码摘要模型学到的模式形式化的方法。

2.1 一个示例

首先，我们介绍一个代码摘要模型和两种输入方法，并以此作为贯穿全文的示例，阐释我们方法的核心思想和总体步骤。

模型。 我们在Java-med和Java-large上使用了当前最先进的代码摘要模型code2seq。与许多其他基于深度神经网络的模型一样，code2seq致力于学习源代码的精确向量表示。这类向量通常称为程序嵌入，通过其数值分量捕捉程序的语义，因此表示相似语义的程序会在向量空间中彼此接近。从高层次来看，code2seq采用生成式方法进行方法名预测。它采用标准的编码器-解码器架构 [Cho 等人，2014；Devlin 等人，2014] 其中，编码器首先将输入方法的抽象语法树嵌入为向量，然后解码器使用这些向量生成由单词组成的序列形式的方法名称（例如，`reverse`和`array` 作为图1中的预测结果）。

输入方法。 图 2a 和 2b 展示了两个名为 `saveBitmapToFile` 的 Java 方法，它们提取自 Java-large 的训练集。该标签的显著特征是 `compress` API，位于 `Bitmap` 类中，两个方法都在阴影框中对其进行了突出显示。值得注意的是，只要输入程序符合其编写语言的语法，code2vec 和 code2seq 都不要求输入程序能够编译。此外，code2vec 和 code2seq 都只接受单个方法作为输入。当输入方法的主体中调用了另一个方法时，不会执行过程间分析，也不会进行内联。给定图



(a) (b)

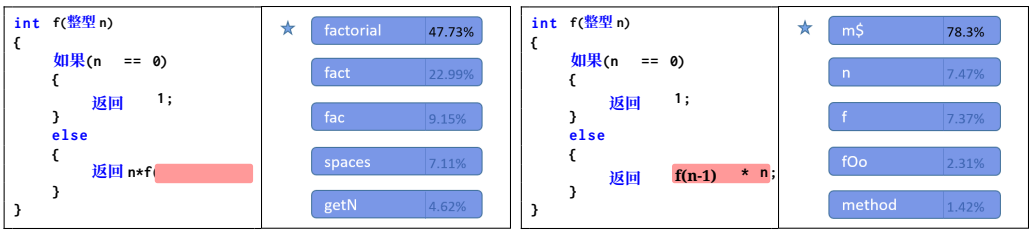
图 2. 两个名为将位图保存到文件的方法。

2a 和 2b 中的这些方法，code2seq 都为它们给出了正确的预测。下文提到 code2vec 或 code2seq 的输入时，我们将程序和方法交替使用。

2.2“抽象、变异、具体化与总结”概述

现在，我们概述一下这项技术。它包含四个主要步骤：抽象、变异、具体化与总结。

2.2.1 抽象。尽管基于深度神经网络的模型在程序领域日益普及，Wang 和 Christodorescu [2019] 提醒说，这些模型的预测结果明显不稳定。简单、自然且能保持语义的变换，往往会导致模型改变预测结果。图3 展示了一个例子：原始方法（3a）被 code2vec 正确预测为阶乘，而变换后的方法（3b）虽然语义等价，却被完全误判。我们只是交换了乘法运算数的顺序，但前五项预测中没有一项与真实值有丝毫相似之处。



(a) 原始方法的预测。 （b）变换后的方法的预测。

图3。一种简单、自然的语义保持变换会导致 code2vec 改变其预测结果。请注意，在变换后的方法上，排名第一的预测的概率甚至更高。

他们的发现表明，模型并不会将注意力均匀分布在方法的整个结构上，而是专注于一个小型的局部代码窗口来进行预测。为了验证这一假设，我们旨在找出模型主要基于其进行预测的代码窗口。一个简单的想法是穷举给定方法中的所有语句子集，找出能够让模型做出与原始程序相同预测的最小子集

。然而，随之而来的是一个挑战：由于需要遍历的子集数量会随着方法规模呈指数级增长，如何才能有效地扩展到由大量语句组成的方法？我们将在第 3.2.1 节中详细讨论如何克服这一挑战。

图 4 展示了我们为图2a 和2b中的方法发现的最小程序。我们想强调三点。首先，（1）两个种子中的语句都没有反映方法的名称，`bos.close()`虽然表明文件操作的概率较高，但并不代表“保存到文件”，也没有将其与“位图”联系起来。其余语句都是完全无关的日志/显示 API。code2seq 仍然为这两个方法预测标签`saveBitmapToFile`的原因在于方法头中提供的参数，因为将类名从`Bitmap`改为`Image`会导致 code2seq 为这两个方法预测出不同的标签。其次，（2）两个种子中都不存在区别性特征（即 压缩 API 位于`Bitmap`类中），这使人们严重怀疑 code2seq 究竟学到了什么。最后，（3）尽管无关，两个种子中的所有语句对于维持 code2seq 的预测都是必要的。这表明 code2seq 会考虑方法的句法结构——在图4a的情况下，即两个连续的方法调用——来进行预测。

```
公有 无返回值 将位图保存到文件(位图 b mp, File 文件) {
    bos.close();
    Log.e(TAG, "保存失败 帧", e);
}
```

(a)

```
公有 无返回值 将位图保存到文件(字符串 文件字符串, 位图 map 位图, 视图容器, 布尔值 是否分享) {
    SnackbarUtil.showShort(容器, "保存到 文件保存成功");
}
```

(b)

图4. (a)/(b) 展示了图2a/2b中该方法的种子。

2.2.2 变异。 众所周知，机器学习模型并非仅仅通过死记硬背建立从输入数据到预测标签的映射，而是从数据中发现能够泛化到具有相似特征的数据上的模式。基于这一认识，我们旨在寻找与种子相似的额外程序，而这些程序无法仅通过纯粹的程序抽象方法获得。对种子进行变异时，我们使用标准的树编辑操作修改其抽象语法树（即节点插入、删除和重命名），并确保生成的变异体同样遵循该语言的句法文法。图5 展示了我们发现的众多变异体中的两个，它们也保留了 code2seq 针对原始程序作出的预测。

```
公有 无返回值 将位图保存到文件(位图 位图, File 文件) {
    _未知_();
    _类型_ _变量_ =
        _未知_("保存失败 帧");
}
```

(a)

```
公有 无返回值 将位图保存到文件(字符串 文件字符串, 位图 map 位图, 视图容器, 布尔值 是否分享) {
    _未知_._未知_(_变量_,
        "保存到 文件成功");
}
```

(b)

图 5。我们发现的两种 变异方式，在众多变异方式中，它们也能保留预测标签 saveBitmapToFile。

我们总结出两点。首先，单个词元的取值会对 code2seq 产生重大影响：在两个变异体中，都必须包含这两个字符串常量，code2seq 才会预测将位图保存到文件。其次，正如上文所述，code2seq 关注方法所呈现的句法结构。例如，在图5b中，即使存在“保存到文件成功”——这是 code2seq 要寻找的模式的主要部分——也必须将其作为参数传入方法

调用。对字符串进行任何其他操作（例如，String _var_= “保存到文件成功”）都会导致 code2seq 预测出不同的标签。

2.2.3 具体化。 尽管 code2seq 的预测主要由种子和变异体决定，但直接将种子和变异体视为 code2seq 为标签“将位图保存到文件”学到的模式定义，是一种有缺陷的方法。因为没有任何形式化保证能够确保：当种子语句或变异体语句周围存在任意代码上下文时，预测标签仍为“将位图保存到文件”。因此，我们首先探索基于种子和变异体的所有完整程序构成的空间，这些程序均为有效具体化结果；随后使用有效具体化结果来定义 code2seq 为标签“将位图保存到文件”学到的模式。当具体化程序保留模型对原始程序所做的预测时，我们称其为有效程序。

要合成一个完整程序，只需枚举所有可以插入种子或变异体中的语句。然而，这种方法很可能无法实现，因为任何编程语言的语法所能枚举的程序空间都是无限的。在 3.2.3 节中，我们将说明如何在寻找有效具体化结果时克服这一可行性难题。图 6 展示了基于变异体的两个有效具体化结果（图 5a 和5b），其语句在阴影框中突出显示。显然，这两个程序在语法乃至控制流结构方面都截然不同，而且无论如何都不表达原始方法的语义。code2seq 之所以保持原有预测，纯粹是因为存在变异体语句。这些例子有力地证明，code2seq 并未学会表示方法的语义；相反，它关注局部的小型句法特征，以此记忆方法名称。

```
公有 无返回值 将位图保存到文件(位图                    b
    File 文件) {                                    mp,
    _未知_();
    执行{循环 (;   _变量_;) {
            final org.jbox2d.common.Vec2
                中心 =新建
                org.jbox2d.common.Vec2(
                上界);
        }
        out.y = (下界.y +
            上界.y) * 0.5F;
    }当 (t1          > t2);
    _类型_ _变量_ =
        _未知_("保存失败                    帧");
}
```

(a)

```
公有 无返回值 将位图保存到文件(字符串          文件字符串,  位图
位图,    视图容器,          布尔值    是否分享) {
    最终 二维向量 p = 参数池.pop二维向量();
    最终 二维向量 d = argPool.popVec2();
    p.set(input.p1);
    d.set(input.p2).subLocal(input.p1);
    如果 (b.下界.x - a.上界.x > 0.0f ||
            b.下界.y - a.上界.y > 0.0f)
        返回;
    否则如果(_变量_)
        avian.UTF-8.cram(buf, j++,
            ((x & 0x1f) << 6) |(y & 0x3f));
    _未知_。_未知_（_变量_,
        "保存到   文件成功");
}
```

(b)

图 6. code2seq 预测为将位图保存到文件的两个具体化程序。

2.2.4 总结。 最后，为了定义 code2seq 为标签 将位图保存到文件 学到的模式，我们推断出一个上下文无关文法，用以描述上一步生成的所有有效具体化结果。事实上，这是一个文法推断问题 [Biermann 和 Feldman, 1972； Stevenson 和 Cordy2014] 其成功在很大程度上仍局限于推断正则文法 [Oncina 和 Garcia1992]。幸运的是，我们的问题设定要简单得多。也就是说，要处理的程序本身已经遵循其所用编程语言的上下文无关文法；换句话说，我们不需要从头生成产生式规则，而是可以重用编译器在解析每个具体化方法时所用的规则。具体而言，我们将描述每个具体化方法的文法取并集，作为 code2seq 为标签将位图保存到文件 学到的模式的定义。

⟨种子声明⟩ ::= ⟨方法头⟩{⟨种子代码块语句⟩?}
⟨方法头⟩ ::= ⟨方法修饰符⟩?⟨返回类型⟩⟨方法声明符⟩⟨抛出声明⟩?
⟨返回类型⟩ ::= 布尔值 | 字节 | 短整型 |整型 | 长整型 | 字符型 | 浮点型 | 双精度浮点型 |无返回值
⟨方法修饰符⟩ ::= ⟨方法修饰符⟩ |⟨方法修饰符⟩⟨方法修饰符⟩
⟨方法修饰符⟩ ::= 公有 |受保护 |私有 | 静态
⟨方法声明符⟩ ::= ⟨标识符⟩(⟨种子参数⟩)
⟨种子参数⟩ ::= 位图 bmp，文件 file
| 字符串 file，位图 bitmap，视图容器 container，布尔值 isShare
⟨种子代码块语句⟩ ::= ⟨代码块语句⟩⟨种子核心⟩⟨代码块语句⟩
⟨种子核心⟩ ::= ⟨种子*1*核心⟩ |⟨种子*2*核心⟩
⟨种子*1*核心⟩ ::= ⟨标识符⟩();⟨代码块语句⟩⟨种子*1*语句⟩
⟨种子*1*语句⟩ ::= 名称不明确(“保存帧失败”);
| ⟨类型⟩⟨标识符⟩ = ⟨名称不明确⟩(“保存帧失败”);
⟨种子*2*核心⟩ ::= 名称不明确(标识符, “保存到文件成功”);
| ⟨名称不明确⟩(⟨标识符⟩, “保存到文件失败”);
⟨名称不明确⟩ ::= ⟨标识符⟩ |⟨标识符⟩.⟨标识符⟩
⟨代码块语句⟩ ::= ⟨代码块语句⟩ |⟨代码块语句⟩⟨代码块语句⟩
⟨代码块语句⟩ ::= ⟨局部变量声明语句⟩ |⟨语句⟩

图 7. 为 saveBitmapToFile 学到的模式 code2seq 所定义的上下文无关语法推断结果。

为清晰起见并简化说明，图 7 展示了仅根据图 5 中两个变异体的具体化结果推断出的上下文无关文法。我们采用巴科斯-诺尔范式的记法，并添加一个额外的量词“？”，表示被量化的终结符或非终结符出现零次或一次。该文法的关键在于描述非终结符 <种子代码块语句> 如何被替换的产生式规则。下文中，对非终结符节点 <种子1核心>和 <种子2核心> 的替换分别指向与两个种子相对应的具体化结果。具体而言，每条产生式规则都描述了如何将后续规则中定义的代码块语句插入每个种子，同时不改变 code2seq 的预测。对于图 7中未定义替换方式的非终结符节点，我们沿用 Java 句法文法中定义的产生式规则；Java 是 code2seq 默认的输入编程语言。

3 方法

本节将详细介绍我们如何形式化代码摘要模型学到的模式。具体而言，我们将介绍“抽象、变异、具体化与总结”方法。

3.1 问题定义

给定一个模型 M，以及一组程序 P（来自 M 的训练集），对于这些程序， M 预测的标签为 L。在这项工作中，我们旨在形式化 M 从 P中学到的模式。换言之，我们的形式化应定义 P 的共同属性， M 将其视为其预测标签为 L的任意程序的“标志”。我们强调两点：（1）P 是从 M的训练集中提取的，而训练集是模型学习所用的数据集的唯一部分。在推理过程中，模型只尝试匹配测试数据中学到的模式，因此我们不考虑非训练程序，以研究模型学到了什么。（2）对于 P 中的某些（甚至全部）程序， M 预测的标签 L 可能不正确；无论如何， M 已经学到了一个可以形式化的模式。事实上，我们提出了一种示例用法：基于 M 做出的错误预测，利用模式定义来提高其准确率。

从高层次来看，我们将代码摘要模型所学习的模式定义为：使用上下文无关文法对所有具有该模式的程序进行概括（定义 3.1）。本节余下部分将说明如何根据一个模型及其训练集推断出这样的上下文无关文法。为了便于阐述，本节将始终使用前文引入的符号。

定义 3.1. (模式) 设 M 为在数据集 D 上训练的模型。设 P **(使得** P $\subsetneq$ D**)** 为一组程序，对于这些程序， M 预测的标签为 L。从 P 中学习到的、用于预测标签L 的模式 M 是一个上下文无关文法 $G = <V, \Sigma, S, R>$，它包含非终结符 V、终结符 Σ、一个开始符号S和产生式规则 R，用于指定 M预测标签 L的程序所共有的属性。

3.2 抽象、变异、具体化与总结算法

本节介绍我们的模式形式化算法。具体而言，本节描述四个关键功能组件：抽象、变异、具体化和总结。

3.2.1 抽象。 此步骤的目标是从每个方法中识别一个片段 P 在 $\mathcal{P}$ 该片段能够捕捉模型对以下对象所做预测的本质 P。我们将这样的片段称为种子（定义 3.2），它们满足充分和必要这两个性质。

定义 3.2。 (种子) 给定一个训练方法 P，其主体由一组语句 B组成；模型 M 预测 P的标签为 L。若另一个方法[1] ϵ 的语句主体为 $\hat{B}$使得 $B \subseteq B$，则称其为 P 的种子当且仅当它是充分的，即 M 也预测 ϵ的标签为 L；并且是 ^必要的，即不存在主体为 B' 的 方法 ϵ' 使得 $B' \subsetneq B$，且 M 也预测 ϵ'的标签为 L。

充分性的直觉在于确保仅凭种子语句就能使模型得出与其对原始方法所作预测相同的预测。至于必要性，我们的定义意味着，种子语句的任何子集都不具备相同的能力。换言之，移除种子中的任意一条语句都会导致模型改变其预测。

定义 3.2 并不能保证一个方法中的种子是唯一的。如果主体中恰好存在多组语句满足充分必要条件，那么一个方法就会有多个种子。事实上，对于图 2a/2b中的程序，我们在图 8a/8b中展示了它的另一个种子。

[1]在处理每个方法时，我们不会更改其方法头，而只更改其主体；因此，我们不会明确说明 ϵ的方法头需要与 P的方法头相同这一要求。后续的定义和定理也同样如此。为简便起见，我们将方法的主体视为由语句构成的集合，其中包括非控制流语句，以及其主体被替换为空代码块的控制流语句（参见 第 5.2节）。

```
公有 无返回值 将位图保存到文件(位图 bmp, File file) {
    尝试 {缓冲输出流 bos = 新建 缓冲输出流(新建 文件输出流(file));}
    捕获(IOException e){
        Log.e(TAG, "保存失败帧", e);}
}
```

(a)

```
公有 无返回值 将位图保存到文件(字符串 文件字符串, 位图 位图,
    视图容器, 布尔值 是否分享) {
    SnackbarUtil.showShort(容器,
        "保存到 file 失败");
}
```

(b)

图 8. 图 2a 和 2b中的程序的另外两个种子。

请注意，多个种子的存在并不一定会违背我们的论断：模型的预测基于一个小型的局部代码窗口。模型的一种常见行为是，以最容易识别的语句为依据，同时从剩余语句中获取足够的信号，从而得出原始预测。图 4a 和 8a 中的种子很好地体现了这种行为。code2seq 将 `Log.e(TAG, “failed to save frame”, e)` 视为特征表示的主导语句，但仍需要其他语句的帮助（例如 `bos.close()` 在图 4a中，以及图 8a中 try-catch 子句内流类的实例化），才能预测标签 `saveBitmapToFile`。换句话说，我们认为，模型所查看的窗口以主导语句为中心，并延伸至各种辅助语句。我们还发现，在一种情况下，模型会查看方法中的不同位置：当多条语句呈现出相似的程序特征时，例如图 4b 和8b 中的两个种子，它们仅在常量字符串中的一个单词上有所不同。因此，从特征表示的角度来看，可以认为模型关注的是程序中同一个代码窗口。

要识别给定方法中的种子，我们可以采用蛮力方法，穷举主体中的所有语句子集。该算法的运行时间为指数时间，并会产生 2^n-2 (其中n 是方法中的语句数；我们无需考虑空种子或方法本身，因此有“-2”个预测）。尽管该方法能够处理规模较小的方法，但随着方法中的语句数增加，其可扩展性会变差。为了解决潜在的可扩展性问题，我们提出了对蛮力方法的一项优化，该优化在平均情况下以二次时间运行。此优化基于模型的单调性性质（定义 3.3）。在给出单调性的形式化定义之前，我们先从较高层面解释其直观含义。由于模型的权重经过优化以拟合训练数据，因此模型在未见数据上的表现有所不同，其差异程度取决于两个数据集所来自的分布以及模型本身的容量。一个具体的证据是：在任何广受认可且成熟的基准中，无论模型有多强大，也无论训练数据与测试数据有多相似，模型在测试集上的准确率总是低于在训练集上的准确率。由于证明这一性质超出了本文的范围，我们假设模型具有单调性；通过大规模实验，我们尚未发现违反这一性质的情况。

定义 3.3。 **(单调性)** 给定模型 M、来自 M 训练集的方法 P_0 、 P_0 的种子 ϵ、不在 M 训练集中的方法 P_n，以及将 P_0 的 AST 转换为 P_n 的 AST 的最短树编辑操作序列δ_1，δ_2，……，δ_n，若对 P_j（其中 $n \geq j > i$）作出相同预测时，对 P_i（P_{i-1} 上应用 δ_i 的结果，其中 $n > i \geq 1$）也会作出与对 P_0 相同的预测，则称其具有单调性当且仅当。类似地，若对 P_j（其中 $i > j \geq 1$）作出不同预测时，对 P_i（其中 $n \geq i > 1$）也会作出与对 P_0 不同的预测，则 M 会对 P_i（其中 $n \geq i > 1$）作出与对 P_0 不同的预测。

算法1 展示了如何在给定的方法中查找种子。关键在于避免在不存在任何种子的语句集合中进行搜索。每当选定的_语句被发现

算法 1： 在给定方法中查找种子

输入： 方法、模型、标签、已遍历的_语句
输出： 种子

```
1  过程 FindSeeds(method，model，label，traversed_statements，seeds)
2      开始
3          大小 ← method.body.大小()
4          对于k ← 1 到 大小 执行
5              已选择的_语句 ← 选择语句(method.body, k, 已遍历的_语句)
6              预测 ← 模型.预测(已选择的_语句, method.head)
7              如果 预测 ==标签 那么
8                  种子 ← 种子 ∩ { new Method(selected_语句, method.head) }
                   // ∖' 表示集合差
9                  剩余_语句 ← 方法.主体 ∖ 已选_语句
                   /* IsAbsent（第19-25行），一种常数时间算法，检查
                       剩余语句中是否存在其他种子 */
10                 如果 IsAbsent(剩余_语句,方法.head,模型,标签) 则
11                     返回
                   /* 当执行到此处时，会发现更多种子，
                       这在第 17-18 */
12                 is_继续 ← True
13                 跳出
14             else
15                 已遍历_语句 ← 已遍历_语句 ∩ { 已选中的_语句 }
16         如果是_继续然后
17             m ←new Method(剩余_语句, 方法.方法头)
18             返回 FindSeeds(m，模型，标签，已遍历_语句，种子)
19 函数IsAbsent(剩余_语句，方法_方法头，模型，标签)
20     开始
21         预测 ← 模型。预测(剩余_语句，方法_方法头)
22         如果预测 != 标签 那么
23             返回 真
24         else
25             返回 假
```

满足充分性条件时，我们便将其保存为种子（第8行），并验证该方法中的剩余语句不存在种子（第10行）。得益于单调性，验证不存在种子只需常数时间。也就是说，如果剩余语句的整体无法使模型得出其对原始方法所作的预测，那么其中任何子集也无法做到，因此无需检查。在这种情况下，算法会返回，其输出可通过种子获取。然而，如果剩余语句中存在更多种子，我们就会跳出当前循环（第13行），并通过递归调用自身来搜索其他种子；调用时使用由剩余语句组成的新方法（第17–18行）。为避免重复尝试

相同的语句，我们会将先前尝试过的内容保存到已遍历的_语句（第15行），以便今后寻找更多种子时跳过这些内容（第5行）。我们不会显式检查必要性，因为所有发现的种子按构造都是最小的。就运行时间而言，算法1 保证通过遍历不超过 $\sum_{k=1}^{m} {}^{n}C_k$组语句找到所有种子，其中 n 是给定方法中的语句数量， m 是该方法中最大种子包含的语句数量，而 ${}^{n}C_k$ 表示从 k 个对象中选取 n的组合数。由于绝大多数种子至多由两条语句组成，算法1 在平均情况下以二次时间运行。

3.2.2 变异。在这一步中，我们着手寻找与种子相似的程序，这些程序也呈现出 M 在预测标签时所查找的模式 L。具体而言，我们对每个种子的抽象语法树进行变异，通过标准的树编辑操作生成新程序：添加节点、删除节点和重命名节点。算法2 详细说明了如何根据定义3.4计算所有变异体的集合。关于定义3.4提出动机的讨论，我们推迟到第3.2.3节。

ˆ 定义 3.4。（变异体）给定一种训练方法 P（其主体为 B）、其种子 ϵ（其主体为 B），
– ˆ 以及一个模型 M，若某方法 ω 的主体 B 是 B 的一个变体（通过对 B 中一条或多条语句的抽象语法树进行树编辑操作得到），则称其为 ϵ 的变异体，当且仅当 M 对 ϵ 和 ω' 给出相同的预测，且 ω'的主体是 $B \setminus B \cup B$。若对 ω 进行任何会使其与 ϵ 距离进一步增大的修改，都会导致 M 预测出的标签与 M 对 ϵ的预测不同，则称 ω 为最弱变异体。相反，若只有 ω 能保留模型对 ϵ 所做的预测（图5中的程序即为例子），则称其为最强变异体。

算法 2 采用迭代方法，通过一组种子查找变异体。在每次迭代中，我们枚举将上一轮发现的变异体转换为其他形式所需的最小编辑（第 5 行）。所谓最小，是指对相应变异体施加更小的编辑会导致语法无效程序（例如，将 a+b 的抽象语法树转换为 a[b] 的抽象语法树，所需的最小树编辑操作数为 2）。在第一轮迭代中，我们转换作为输入给定的种子。请注意，在为每个变异体枚举编辑时，我们只考虑修改，因为根据种子的定义，删除必然会产生无效变异体；插入本质上与种子修改无关，将在下一步处理。随后，我们根据变异体的定义检查每个转换后程序的有效性（第 7-10 行）；若转换后的程序通过有效性检查，我们便将其保存为变异体，并为后续迭代中的进一步修改做好准备（第 11-12 行）。另一方面，若转换后的程序未通过检查，则会被永久丢弃。根据单调性，进一步修改该程序不会产生有效变异体。当没有任何转换后的程序通过有效性检查时，迭代便会停止，此时我们已找到所有变异体。

更改标识符和常量。关于标识符终端节点的重命名操作（例如，变量、类型或方法名称），可以将它们更改为任何符合词法语法的值（例如，语言保留的关键字不能用于命名变量）。为充分探索搜索空间，我们既考虑在给定方法内交换标识符，也考虑将它们更改为未出现在该方法中的单词。对于常量，我们采用相同的方法，同时避免产生类型错误（例如，整型 a = ‘3’）。

3.2.3 具体化。 尽管种子和变异体在很大程度上决定了模型做出的预测，但将它们的属性视为模型所学模式的定义并不妥当。由于种子（或变异体）中的语句无法保证保留模型对原始程序所做的预测，考虑到它们周围的上下文可以由任意代码构成

算法 2： 通过一组种子查找变异体

输入： 方法、模型、标签、种子
输出： 变异体

```
过程 FindMutants(方法，模型，标签，种子，变异体)
    开始
        最新_变异体 ← ∅
        对每个种子 ∈ 种子 执行
            转换后的_种子 ← EnumerateMinimalValidEdits(种子)
            对每个 转换后的_种子 ∈转换后的_种子 执行
                剩余_语句 ← 方法 \ 种子
                转换后的_种子.主体 ← 转换后的_种子.主体 ∩ 剩余_语句
                预测 ← 模型.预测(转换后的_种子)
                如果 预测 ==标签 那么
                    最新_变异体 ← 最新_变异体 ∩ { 转换后的_种子 }
                    变异体 ← 变异体 ∩ { 已转换的_种子 }
        如果 最新_变异体 != ∅ 那么
            FindMutants(方法，模型，标签，最新_变异体，变异体)
```

。因此，这一步的目标是找出由种子（或变异体）语句锚定的程序空间，并确保模型对原始程序所做的预测得以保留。

基于种子具体化。 如第 3.2.1节所述，由于模型的训练机制本身如此，它们在训练数据和测试数据上的表现差异不可避免。未见样本与训练示例越接近，它被赋予与训练示例相同预测标签的概率就越高。基于这一特性，我们在下文介绍将种子具体化的技术。

我们只需在原始方法中恢复已删除的语句，然后逐步编辑这些恢复的语句（同时保持种子语句不变），直到编辑后的结果方法不再保留模型对种子所作的预测。我们的直觉是，使用一组闭区间来量化所有可能的具体化结果所构成的空间，每个区间都衡量具体化后的方法沿编辑轨迹与原始方法相距多远[2]。每个区间的高端是距离最近的具体化方法（即原始方法本身），它为可能的修改留出了最大的空间。低端则是距离最远的具体化方法（称为阈值方法）；尽管仍以种子语句为基础，但它对非种子语句所作的累积编辑已耗尽预算。换句话说，任何使其与原始方法距离进一步拉大的改动，都会导致模型改变对种子所作的预测。根据单调性，位于每个区间两端之间的任何程序都是有效的具体化结果。回顾第2节说明性示例中推导出的上下文无关文法，为了使说明更清晰，我们没有限制产生式规则的递归深度。实际上，将递归深度设为三，对于图4 和5中的种子和变异体的具体化而言，能够很好地近似每个区间的低端。最后，作为一种验证机制，我们将每个具体化方法交由模型处理，以确认其有效性

[2]如算法 2中将种子变异为变异体一样，在种子具体化过程中，改变非种子语句也存在多种编辑轨迹。

并获取模型预测的标签。只有当具体化方法使模型得出其对原始程序所作的预测时，我们才会保留该方法。

具体而言，在改变非种子语句时，我们首先保留原始方法的控制流结构，仅修改控制结构中的非控制语句或表达式。随后，在切换到不同的控制流结构时，我们会将每个区间的上界重置为 Q，即在所有采用新控制流结构的程序中，最接近 P 的程序 v。形式化地说，我们在方程 Q 中定义1：

$$
\begin{aligned}
Q &= \underset{W \in \mathcal{W}}{\operatorname{argmin}} |P - W|^T \\
\mathcal{W} &= \{W \mid CF(W) = v, NCF(W) \subseteq NCF(P) \cap \{\varnothing_s\}\}
\end{aligned} \tag{1}
$$

其中， P 是生成种子的原始程序。$CF(W)$ 返回 W 的控制流结构。$NCF(W)$ 返回 W 的非控制语句集合。我们考虑 $\{\varnothing_s\}$（一个空语句集合），是为了在现有非控制语句用尽时，允许控制语句具有空代码体（例如，控制语句的数量大于非控制语句的数量）。类似地，我们施加的余量 Q 会随着应用于 Q 的更改而逐渐消耗。与之前一样，当 Q 不再保留为种子建立的预测模型时，枚举过程终止；我们将每个区间两端之间的程序视为具有不同控制流结构的种子的具体化。

通过变异体进行具体化。在上述方法中，种子语句充当方法中的锚点，并在整个具体化过程中保持不变。现在，我们来讨论种子语句被修改时该如何处理。一旦种子被修改，即使原始方法中的非种子语句提供了帮助，这些语句也可能无法维持原预测模型所做出的预测。图9 给出了一个例子：尽管所有非种子语句都存在，但由于种子语句被修改，两个程序都改变了 code2seq 的预测。对于这类经过修改、即使存在非种子语句也无法保留原预测模型所做预测的种子，我们不会在后续具体化中考虑它们。因为根据单调性，它们没有足够的变更预算来进一步修改，以具体化这些种子。另一方面，剩余的经过修改的种子恰好就是变异体（定义3.4）；其中，变更预算最少的称为最弱变异体。

```
公有 无返回值 保存位图(位图 b mp, File 文件) {
  尝试 {
      缓冲输出流 bos = 新建 BufferedOut-
          putStream(new 文件输出流(file);
      bmp.compress(位图.位图压缩格式 PNG ,
          90 , bos);

      bos.close();
      Log.e("成功");
  }
  catch (输入输出异常 e) {
      Log.e(TAG, "失败", e);
  }

}
```

(a)

```
公有 无返回值 保存位图(字符串 文件字符串, 位图 位图,
  视图容器, 布尔值 是否分享) {
  File 文件 =新建 文件(文件字符串);
  如果(是否分享 && 文件.exists()) 返回;
  尝试{ 文件输出流 fos =新建 文件输出流(file);
      布尔值 isCompress =bitmap.compress(Bitmap.
          PNG 压缩格式 ,90, fos);
      如果(isCompress)
          SnackbarUtil.showShort(容器 ,
              "保存成功");
      else
          SnackbarUtil.showShort(容器 ,
              "保存 失败");
      fos.flush(); fos.close(); }
  捕获 (输入输出异常 e) { e.printStackTrace (); }
}
```

(b)

图9. 修改种子语句会导致 code2seq 改变其预测，尽管所有非种子语句都存在。浅色方框标出种子语句中的改动。

为了将每个变异体具体化为一个完整的方法，我们采用与具体化种子相同的流程。也就是说，我们将原始方法中的非种子语句注入变异体的主体。结果方法具有最大的潜在变更空间，并会持续被修改，直到模型不再保留其对种子所做的预测。与种子具体化类似，变异体中的语句不会被修改，因为变异体本身已经涵盖了所有有效变更。此外，我们还采用验证机制，确保每个基于变异体的具体化结果都有效。

3.2.4 总结。 最后，我们推断出一个上下文无关文法，用于精确描述上一步生成的所有具体化方法。我们将此文法声明为从 PM 中学习到的、关于标签 L的模式定义。为了解决这一文法推断问题，我们不重新发明轮子，而是复用输入程序已经采用的句法文法。也就是说，我们提取编译器在解析每个具体化方法时所使用的产生式规则，然后将它们组合成统一的文法。形式化地说，我们取从每个文法中提取的终结符、非终结符和产生式规则的并集。关于 code2seq 为标签 saveBitmapToFile 学习到的模式定义，我们推断出的文法描述了 13 种不同的控制流结构，并且可以实例化为数百个具有不同句法结构的程序。换句话说，当枚举出的程序与原始方法的大小相近时，种子和变异体大多数情况下都能够保留预测标签 saveBitmapToFile。通过推断出的文法，我们得出结论：code2seq 并未为名为 saveBitmapToFile 的方法学习语义表示，而是通过少量句法特征记住了这些方法。

3.3 正确性

我们证明，我们的“抽象、变异、具体化与总结”算法符合 定义 3.1（模式）。具体而言，定理 3.1 和3.2 为我们的算法给出了可靠性和完备性的证明。

定理 3.1 （可靠性）。给定一个基于数据集 D 训练的模型 M，以及算法推断出的语法 G，该语法用于定义从 D 中学到的、用于预测标签 L的模式 M，当且仅当 G 没有描述任何一个 M 无法预测标签 L的程序时，称该算法是可靠的。

证明。种子或变异体的所有具体化结果都通过 M的预测进行验证。算法推断出的上下文无关语法精确描述了所有有效具体化。因此，根据构造，定理成立。□

定理 3.2 （完备性）。给定一个基于数据集 D 训练的模型 M，以及算法推断出的语法 G，该语法用于定义从 D 中学到的、用于预测标签 L的模式 M，当且仅当 G 描述了任何一个 M 能够预测标签 L的程序时，称该算法是完备的。

证明。反设存在一个程序 σ，使得 M 将标签 L 预测为适用于 σ，而 G并未描述 σ。关于 σ的属性，以下条件之一必须成立：

(a) σ 来自 M的训练集。回顾我们算法中的具体化步骤：该步骤使用所有预测标签为 L 的训练程序来枚举有效具体化。换言之，所有训练程序都已包含在具体化步骤中，并且都会由 G描述。这与假设矛盾。因此，该条件不成立。

(b) σ 不来自训练 g 集 M。

（1）σ 包含源自训练程序 P 的种子 ϵ 中的语句。根据我们的假设， G 并不描述 σ，因此 σ 不是 ϵ 的有效具体化，这意味着 σ 是

任何将 P 更改为阈值方法的编辑轨迹都不会经过的。根据单调性，M不会为 σ预测标签 L，从而与该假设相矛盾。（2） σ 包含源自种子的变异体中的语句。采用与（1）中相同的策略，也可以推翻这一条件。（3） σ 既不包含种子中的语句，也不包含变异体中的语句。如具体化步骤中所述，σ 不会使 M 为 L预测标签。该条件同样为假。

Sinc如果上述条件均不满足，则可以推断 σ 不存在 t. □

4 模式定义的应用示例

本节将展示模式定义的实际意义：一种评估鲁棒性的新方法，以及一种提高代码摘要模型准确率的新技术。

4.1 评估代码摘要模型的鲁棒性

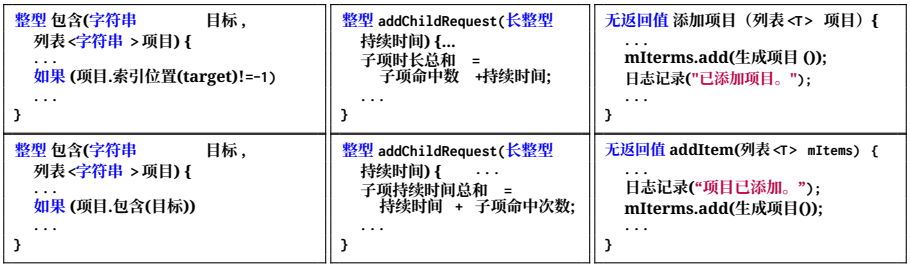
模型的鲁棒性指其预测的可靠性，尤其是面对对抗样本时。对抗样本是一种特殊类型的数据，通过系统性地扰动模型输入而生成。Szegedy 等人 [2013] 最早在图像分类领域发现了对抗样本的存在：视觉上无法区分的扰动会导致模型改变其对原始图像所作的预测。现有的模型鲁棒性评估方法可分为两类：正式验证一个下界 B_l [Gehr 等人 2018; Raghunathan 等人 2018; Wang 等人 2018] (即保证在不超过 B_l 相对于某种距离度量（例如 L_0， L_1 或 L_∞）的扰动下，预测仍然成立；另一类则是构造攻击，以证明一个上界B_u [Carlini 和 Wagner 2017; Goodfellow 等人 2015] (即不小于 B_u 的扰动足以使模型改变其预测）。

Wang 和 Christodorescu 的方法属于后者，即通过对原始程序应用语义保持变换来生成对抗样本。感兴趣的读者可以查阅补充材料，了解这些变换的示例。尽管取得了重要发现，他们的方法仍存在两个问题。首先，由于可应用的变换很多，而且每种变换都可以应用于给定方法中的多个位置，盲目尝试所有可能性效率很低。此外，他们的变换往往会对原始方法造成相当大的改动，因此倾向于找到较宽松的界限，无法准确反映模型的鲁棒性。

为解决他们方法的不足，我们利用模式定义来精准定位对抗样本，从而得到比 Wang 和 Christodorescu 的方法紧得多的界限。直观而言，由于我们的目标是在扰动输入方法时尽量减少改动，因此我们只修改种子语句——也就是模型会重点关注的方法部分——以生成对抗样本。我们引入了四种语义保持变换——变量重命名、操作数交换、API 替换和语句重排序——它们只对输入方法的种子部分做少量修改。图 10 展示了除变量重命名之外的每种变换的示例。每一列中，上方的代码块是原始方法，下方的代码块是变换后的方法。与 Wang 和 Christodorescu 的方法相比，我们的变换对原始程序的改动更小。

4.2 提高代码摘要模型的准确率

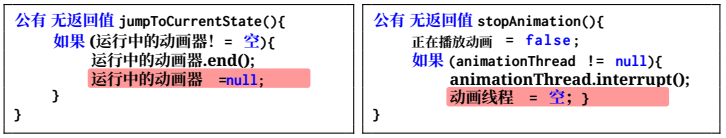
我们提出了一种基于模式定义的新技术，可提高代码摘要模型的预测准确率。该技术建立在我们对



(a) API 替换 (b) 操作数交换 (c) 语句重排序

图10。三种保持语义的程序变换。上排展示原始方法，下排展示变换后的方法。

模型所做的错误预测的一项重要观察之上。当模型错误地将某个方法预测为具有标签 *A* 的标签 *B*时，该方法的种子往往与另一个具有标签 *B*的方法的种子相似。在这种情况下，模型无法清楚地区分两者，从而导致错误预测。图 11 展示了一个例子：两个方法的种子相似，但名称不同。code2seq 将图 11a 中的方法错误预测为`stopAnimation`，也就是图 11b中方法的名称。



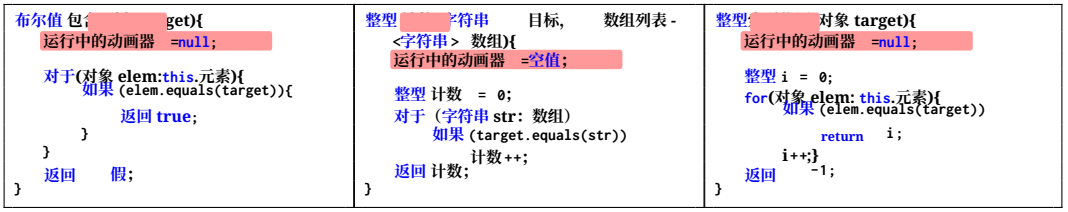
(a) (b)

图 11. code2seq 错误地将（a）的名称预测为 `stopAnimation`，即（b）中方法的名称。原因是（a）中的种子语句与（b）中的种子语句相似。我们在阴影框中突出显示了两个程序的种子语句。

受这一发现启发，我们提出了一种沿用对抗训练思路的新方法 [Good- fellow 等人 2015], ，该方法引导模型在预测标签时关注输入方法的不同部分。我们希望新种子不会与从整个数据集的训练程序中提取的任何现有种子冲突。在技术层面上，我们通过将一个错误预测方法的种子注入多种（即 具有不同标签的）正确预测的方法中，来创建新的训练程序。每个新创建的程序都会被赋予恰好包含该种子的正确预测方法的标签。我们的目的是将导致模型错误预测的旧种子替换为新种子，希望新种子能引导模型得出真实值。

图12展示了一个示例：我们将预测错误的程序（图11a）中的种子语句分别放入三个名为包含、计数和索引位置的方法中。生成的方法会保留其所属方法的名称，如图中所示。在实践中，许多此类实例都会被加入训练集。因此，模型不得不将注意力转向预测错误的程序，因为继续使用运行中的动画器 = 空值作为种子，会影响它们在额外训练样本上的准确率，尽管所属方法的权重更高。

由于种子语句总会发出强烈信号来影响模型，因此不会被忽略；唯一的出路就是消除它们的影响，而这正是重新训练的目标。



(a) (b) (c)

图 12. 用于重新训练代码摘要模型的其他程序。

为选择预测错误的程序，我们针对模型错误率最高的标签，以避免模型对少数离群值过拟合，而忽略了原本能够完美预测的标签。因为对于大量具有相同标签的程序，预测准确率较低表明存在欠拟合问题，而这正是我们新技术所要解决的。

5 评估

我们将“抽象、变异、具体化和总结”这一算法实现为一个名为 PATIC 的工具，它使用上下文无关文法对代码摘要模型所学习的模式进行形式化。在评估的第一部分，我们详细介绍 PATIC 生成的模式定义。对于模式定义的示例应用，我们还评估了该方法在发现对抗样本方面的有效性，以及用于提高代码摘要模型预测准确率的新技术。

5.1 评估对象

模型。 code2vec、code2seq、序列 GNN 和 LiGER 是文献中最著名的代码摘要模型。理想情况下，它们都应纳入我们的实验。然而，我们在复现序列 GNN 的结果时遇到了很大困难 [Fernandes 等人2019]。在他们实验中使用的同一基准上，我们的重新实现效果明显较差（即 F1 值低了 10% 以上）。我们怀疑这种差异是由不一致的提取器造成的，这些提取器会将方法转换为适用于该模型的图表示，因为原始提取器是他们整个流程中唯一未开源的部分。[3]考虑到基于效果较差的重新实现所得结果的有效性存疑，我们遗憾地将序列 GNN 排除在实验之外。我们也没有选择 LiGER，这是一种严重依赖程序执行的模型，因为它的适用范围和通用性相当有限。与甚至不要求程序能够编译的 code2vec 和 code2seq 相比，LiGER 要求程序能够执行。因此，在 Java-med 和 Java-large 中，LiGER 仅对不到 10% 的方法进行评估，因为绝大多数方法都无法触发 LiGER 用于学习的有意义执行。

数据集。 我们使用 Java-small、Java-med 和 Java-large，这三个被许多代码摘要模型用于评估的公共数据集。它们由 Alon 等人 [2019提出], , 它们是

[3]我们已努力与序列 GNN 的作者协商解决这一问题。遗憾的是，我们未能在投稿截止日期前找到解决方案。我们已将邮件往来附在补充材料中，供审稿人查阅。

表 1. 比较 code2vec 和 code2seq 的重新实现版本与原始版本。

模型	Java-小型			Java-med			Java-large		
	精确率	召回率	F1	精确率	召回率	F1	精确率	召回率	F1
code2vec	18.51	18.74	18.62	38.12	28.31	32.49	48.15	38.40	42.73
code2vec（重新实现）	**19.23**	**17.72**	**18.44**	**40.32**	**28.89**	**33.66**	**48.90**	**37.26**	**42.29**
code2seq	50.64	37.40	43.02	61.24	47.07	53.23	64.03	55.02	59.19
code2seq（重新实现）	**48.72**	**35.46**	**41.05**	**61.91**	**46.38**	**53.03**	**63.78**	**54.41**	**58.72**

从 GitHub 上大量项目中提取的 Java 方法集合。我们使用在 GitHub 上开源的实现，重新训练了 code2vec 和 code2seq。表 1显示，重新训练的模型与原始模型具有可比性 [Alon 等人，2019a]。

5.2 code2vec 和 code2seq 学到了什么

现在，我们详细介绍 PATIC 为 code2vec 和 code2seq 生成的模式定义。

寻找种子。 总的来说，我们遵循算法 1 来识别给定程序的种子。对于控制语句，我们将其主体视为独立于控制谓词。例如，在抽象一个 `if` 语句时，我们可以删除 `if` 条件并保留主体中的语句，也可以从主体中移除一条非控制语句。我们递归地应用这种方法来处理嵌套控制结构。表 2 展示了 code2vec 和 code2seq 的种子大小，以其包含的词元数量表示（例如均值和中位数）。括号中的数字表示种子的词元数量占整个方法词元数量的百分比。同样，表3 给出了最强变异体的统计数据。显然，这两个模型只学到了局部句法程序特征，因为这些种子未能捕捉输入方法的全局语义属性。

表2。种子的大小。

模型	Java-小型		Java-med		Java-large	
	Mean	中位数	Mean	中位数	Mean	中位数
code2vec	14.23 (24%)	10.00 (19%)	13.13 (18%)	9.00 (12%)	14.30 (21%)	10.00 (21%)
code2seq	11.53 (23%)	9.00 (15%)	14.95 (21%)	11.00 (14%)	18.72 (19%)	16.00 (13%)

表3. 最强变异体的大小。

模型	Java-小型		Java-med		Java-大型	
	Mean	中位数	Mean	中位数	Mean	中位数
code2vec	13.91 (23%)	10.00 (19%)	12.21 (17%)	9.00 (11%)	13.70 (20%)	10.00 (21%)
code2seq	12.03 (24%)	9.00 (15%)	14.32 (21%)	11.00 (13%)	17.33 (18%)	16.00 (13%)

种子与变异体具体化。在具体化步骤中，我们采用与变异步骤相同的方法来处理标识符（例如，变量、类型或方法名称）。一般来说，种子或变异体的具体化结果涵盖了广泛的程序结构，其中许多与种子之外语句的语义并不相似。图13 展示了另外两个具体化结果，除此之外还有图6中的结果。

表 4 展示了一个具体化中包含的不同控制流结构数量。表 5则给出了关于 实际程序实例在句法变体方面的相同统计数据。显然，两个模型对输入方法中的非种子语句几乎完全不予关注，因为这些语句经常被替换为其他截然不同的语句，而不会改变模型对原始程序所作的预测。

表4. 不同控制流结构的数量 res

模型	Java-小型		Java-med		Java-大型	
	Mean	中位数	Mean	中位数	Mean	中位数
code2vec	12.31	10	12.79	10	9.00	8
code2seq	12.22	9	10.43	8	8.49	6

表 5。语法变体的数量

模型	Java-小型		Java-med		Java-large	
	Mean	中位数	Mean	中位数	Mean	中位数
code2vec	246.52	273	248.17	260	336.00	336
code2seq	260.49	275	218.46	230	207.92	225

```
公有 无返回值 将位图保存到文件(位图 b mp, File 文件) {
    当 (A){
        如果 ((x & 0xf0) == 0xe0){
            if (!是否为多字节) {
                缓冲区 = avian.Utf8.widen(buf, j,
                    长度 - 2);
                是否为多字节 = true;
            }
            如果 (A){
                如果 (!(data 是 byte 的实例[]))
                    返回;
            }
        }
    }
    尝试 {缓冲输出流 bos = 新建 BufferedOut-
        putStream(新建 文件输出流(file));}
    捕获(输入输出异常 e) {
        Log.e(TAG, "保存失败 帧", e);}
}
```

(a)

```
公有 无返回值 将位图保存到文件(字符串 文件字符串, 位图
    位图, 视图容器, 布尔值 是否分享) {
    SnackbarUtil.showShort(_变量_,
        "保存 to file 失败");
    如果 (_变量_){
        如果 (_变量_){
            循环 (; _var_;) i++;
            返回 avian.Utf8.修剪(buf, j);
        }
    如果 (!(data 是字节的实例[]))
        返回;
    字节[] b = ((字节[]) (data));
    循环 (整型 i = 0; i <b.length; ++i) {
        if ((((整型) (b[i])) & 0x80) != 0)
            返回;
    }
    返回;
    }
}
```

(b)

图13. 图 8 中种子的两个具体化程序。

code2vec 和 code2seq 主要基于输入方法中很小一部分内容作出预测。

5.3 评估 code2vec 和 code2seq 的鲁棒性

我们基于 PATIC 生成的模式定义，构造针对 code2vec 和 code2seq 的攻击。给定一个输入方法，我们识别其种子语句，并对这些语句应用上述语义保持变换，以寻找潜在的对抗样本。如果一个变换后的程序使模型给出不同于其对原始方法所作预测的结果，则找到了一个对抗样本；模型的鲁棒性可通过对测试集中被正确预测的原始方法及其最近对抗样本之间的距离取平均值来计算。我们采用树编辑距离（包含节点交换操作）作为衡量程序之间距离的指标，因为通常用于对抗学习设置的其他指标，如 L_0、 L_1 或 L_∞，并不适用。表 6 和 7 展示了基于可编译程序的 code2vec 和 code2seq 鲁棒性分数，这可被视为第一道防线。表 8 和 9 展示了两种方法平均需要尝试多少次才能找到最近的对抗样本。对于多个适用的变换，我们按照应用变换后所得程序与原始方法之间距离的升序对其进行排序。对于产生相同距离程序的变换，将随机选择。基线是指 Wang 和 Christodorescu 的方法；与我们的方法相比，该方法需要进行更多尝试才能找到对抗样本。

表6. code2vec 的鲁棒性分数。

方法	小型 Java	中型 Java	Java-large
基线	3.24	2.95	3.31
PATIC	**1.78**	**1.26**	**2.43**

表7. code2seq 的鲁棒性分数。

方法	小型 Java	中型 Java	Java-large
基线	2.39	3.50	3.14
PATIC	**1.12**	**1.68**	**3.02**

我们还进一步深入分析了所获得的鲁棒性分数，发现这两种方法都严重依赖变量重命名来生成对抗样本。为公平评估其他程序变换，我们排除变量重命名，并重复进行相同的实验。表10 和11 分别报告了对于 code2vec 和 code2seq，无法通过其他变换生成对抗样本的程序百分比。利用其余

表8. 找到距离 code2vec 最近的对抗样本所需的平均尝试次数。

方法	小型 Java	中型 Java	大型 Java
基线	12.1	14.3	15.7
PATIC	**2.1**	**3.2**	**2.4**

表9. 为 code2seq 找到最近的对抗样本平均需要的尝试次数。

方法	小型 Java	中型 Java	Java-large
基线	15.4	16.9	22.5
PATIC	**2.6**	**2.4**	**3.7**

能够生成对抗样本的程序，我们在表12 和13中报告了 code2vec 和 code2seq 的鲁棒性分数。表10-13中呈现的结果表明，如果不考虑变量重命名变换，我们的方法不仅能在所有数据集上为更多的程序找到对抗样本，而且对原始方法的修改也显著更小。

表10. 无法为 code2vec 创建对抗样本的程序百分比。

方法	Java-small	Java-med	Java-large
基线	72%	68%	64%
PATIC	**35%**	**27%**	**31%**

表11. 无法为 code2seq 创建对抗样例的程序所占百分比。

方法	Java-small	Java-med	Java-large
基线	68%	64%	67%
PATIC	**28%**	**29%**	**33%**

表12。重新评估 code2vec 的鲁棒性。

方法	小型 Java	中型 Java	Java-large
基线	12.8	9.40	15.1
PATIC	**1.40**	**1.71**	**1.88**

表13。重新评估 code2seq 的鲁棒性。

方法	小型 Java	中型 Java	Java-large
基线	10.3	14.6	13.4
PATIC	**1.69**	**1.37**	**2.58**

code2vec 和 code2seq 都容易受到对抗样本的影响。对给定方法中的种子语句进行扰动，很容易导致 code2vec 和 code2seq 改变其预测。

5.4 提高 code2vec 和 code2seq 的准确率

我们提出了一种新技术，通过引导 code2vec 和 code2seq 纠正自身的错误预测，来提高它们的预测准确率。为 code2vec 和 code2seq 的重新训练做准备，表 14 和15 给出了我们选取错误预测程序进行修正时所涉及的标签数量、模型在这些标签上的平均错误率，以及用于重新训练的生成程序数量。

表 14. code2vec 重新训练的准备工作。

Mis-预测	Java-小型	Java -med	Java-大型
标签数量	73	442	1186
平均错误率	52%	60%	70%
生成的程序数量	6,161	37,774	97,061

表15. code2seq 重新训练的准备工作。

Mis-预测	Java-小型	Java -med	Java-大型
标签数量	118	552	1137
平均错误率	64%	52%	72%
# 生成的程序	6,438	39,592	99,566

表 16 展示了重新训练后 code2vec 和 code2seq 在全部三个数据集上的结果。显然，我们的技术并不只是让模型对其训练集过拟合，因为它们在测试集上的准确率也得到了持续提升，尤其是 code2seq，如今在 Java-med 和 Java-large 上取得了最先进的结果。另一方面，我们承认这一改进并不显著；尽管如此，我们仍然认为，从以下几个方面来看，该技术仍是一项重要贡献。

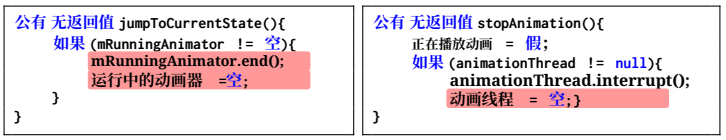
- **可靠性：** 表 16 中展示的所有结果，除了基于 code2seq 和 Java-large 的结果外，均为十次独立训练实例的平均值。由于训练时间超过一周，基于 code2seq 和 Java-large 的结果取五次平均。因此，这两个模型所表现出的更高准确率并非随机噪声，而是可靠的提升。

表16. code2vec 和 code2seq 重新训练后的结果。

模型	Java-小型			Java-中型			Java-大型		
	精确率	召回率	F1	精确率	召回率	F1	精确率	召回率	F1
基线	18.82	17.14	17.94	40.95	28.63	33.70	47.46	37.19	41.70
code2vec	19.23	17.72	18.44	40.32	28.89	33.66	48.90	37.26	42.29
code2vec（重新训练）	**22.75**	**18.59**	**20.46**	**43.47**	**31.48**	**36.51**	**49.35**	**38.58**	**43.31**
基线	49.16	35.25	41.16	62.24	46.07	52.95	63.08	54.12	58.26
code2seq	48.72	35.46	41.05	61.91	46.38	53.03	63.78	54.41	58.72
code2seq（重新训练）	**48.05**	**40.61**	**42.08**	**63.24**	**47.89**	**54.50**	**66.49**	**56.43**	**61.05**

- **简洁性：** 作为一项主要优势，我们的技术本质上是一种数据增强方法，无需更改现有模型的架构。通过根据先前的错误预测，系统地扩充训练集，便能以远低于设计新模型的成本实现性能提升。

- **效果：** 表 16 中的基线使用从 GitHub 随机选取的等量额外数据扩充训练集。新增数据的标签与用于重新训练的生成的程序相同。如表中所示，随机扩充训练集并不总能提高模型的准确率，因为两个模型的表现几乎与之前相同。此外，我们还提供了重新训练对 code2seq 影响的具体证据。对于误预测的程序（图 11a），code2seq 通过扩展种子语句（图 14a中突出显示）纠正了自身的错误，从而能够区分图 14a中的方法与图 14b中的方法。



(a) (b)

图 14。 鉴于（b）中的种子语句更加不同，code2seq 能正确预测（a）的标签。

这种增强方法通过引导 code2vec 和 code2seq 纠正自身的错误预测，提高了两者的准确率；尤其是，code2seq 在现有架构的基础上，在 Java-med 和 Java-large 上取得了最先进的结果。

5.5 关于有效性威胁的讨论

我们的方法面临的一个主要有效性威胁，是我们对代码摘要模型的单调性所作的假设。直观而言，鉴于机器学习模型经过训练以拟合训练数据，单调性是一个合理的假设。此外，我们的大量实验也有力支持了这一行为，因为尚未发现任何违例。不过，我们尚未对这一性质给出原理证明，而该性质对于所评估的模型未必成立。但即便单调性被证明是错误的，我们仍认为我们的方法在很大程度上是有效的。

首先，由于种子、变异体和具体化结果的有效性均通过受测模型所作的预测进行了验证，因此该方法的可靠性不会受到影响。这意味着，我们的主要发现——code2vec 和 code2seq 为预测方法名称而学习到的模式——仍然成立。因此，我们关于模式定义应用的次要贡献——一种评估鲁棒性的新方法，以及一种提高 code2vec 和 code2seq 准确率的新技术——也同样成立。受到影响的方面

是我们方法的完备性，也就是说，定义模型针对某个标签学到了什么的推断语法，很可能会遗漏一些模型同样预测为该标签的程序。不过，鉴于可靠性得以维持及其带来的影响，我们认为，即使没有单调性，该方法仍基本有效。

6 相关工作

本节综述两类相关工作：研究模型学到了什么，以及根据方法的代码主体预测方法名称。

6.1 模型学到的模式

在计算机视觉领域，Han andZhu [2008] 提出了一种属性图文法，用于解析包含人工构造对象的图像，例如建筑物、走廊和厨房。他们的算法专注于仅检测矩形的形状，并使用六条产生式规则描述所检测矩形之间的各种空间关系，以此作为目标检测的基础。随后，类似的语法被应用于布料建模这一人体识别和跟踪中的重要任务，以建模布料构型的广泛变化 [Chen 等人 2006]。他们方法的一项突出优势，是能够根据推理过程中产生式规则的激活情况，深入了解算法所识别的模式。

Zeiler and Fergus [2014] 提出了一种可视化技术，用于揭示中间特征层的功能以及卷积神经网络的运行机制。该技术建立在反卷积网络 [Zeiler 等人 2011], 的基础上，能够展示模型任意层中激发各个特征图的输入刺激。它还可以让人观察训练过程中特征的演变，并诊断模型的潜在问题。作为该技术的另一项贡献，他们能够揭示输入图像中对分类具有重要作用的部分。

Bau 等人 [2019] 提出了一种分析框架，用于可视化和理解生成对抗网络。首先，他们识别出某一层中其特征图对应于检测某类对象的单元（例如树木）。其次，他们在网络中进行干预，以关闭（或重新开启）对这类对象的检测（例如，强制将已识别单元的激活值设为零），并量化消融的平均因果效应。最后，他们考察这些具有因果作用的对象单元与背景之间的上下文关系。

6.2 代码摘要模型

code2vec 是首个应用于跨项目设置的代码摘要模型。其工作方式为：（1）将输入方法的抽象语法树（AST）分解为一组 AST 路径，每条路径都是 AST 中节点之间的一条路径，从一个终结符出发，到达另一个终结符，并经过这两个终结符的共同祖先；（2）聚合为每条 AST 路径学习到的嵌入；（3）根据聚合后的嵌入，对给定的一组标签预测概率分布。code2vec 还采用了注意力机制 [Bahdanau 等人，2015；Vaswani 等人 2017] 为 AST 路径分配不同的权重。换言之，方法的嵌入是 AST 中各路径嵌入的加权和。

code2seq 是文献中另一个值得关注的代码摘要模型。与本质上属于判别模型的 code2vec 不同，code2seq 采用编码器-解码器架构 [Cho 等人，2014；Devlin 等人，2014] 用于将方法名称预测为单词序列。在解码时，code2seq 也会关注一组表示，这些表示旨在结合每条路径及其词元表示，以生成方法名称。随后，code2seq 在 Java-small、Java-med 和 Java-large 这三个 Java 方法公共数据集上均取得了最先进的结果。

与 code2seq 类似，序列 GNN 也采用编码器-解码器架构来预测方法名称。为了对给定的方法进行编码，序列 GNN 依赖于序列模型与图模型的协同

工作。也就是说，循环神经网络首先学习程序中每个标记的序列表示，然后门控图神经网络计算抽象语法树中每个节点的状态。与 code2seq 一样，它们使用另一个循环神经网络将方法名称生成为单词序列。

LiGER 是首个将动态程序特征融入代码摘要学习过程的模型。其核心见解是，执行结果能够提供程序行为直接、精确且规范化的表示，帮助模型超越句法特征进行泛化。另一方面，LiGER 使用从源代码中学习到的符号特征，以减少动态模型对程序执行的严重依赖，因为高覆盖率的执行结果并不总是易于获得。

7 结论

在本文中，我们提出了代码摘要模型所学习模式的首个形式化定义。基于该定义，我们开发了一种用于生成此类模式定义的可靠算法，并实现了一个可用于形式化 code2vec 和 code2seq 所学习模式的系统。我们发现，code2vec 和 code2seq 都高度关注输入方法中一个局部且很小的片段来预测其名称，这表明两个模型对于全局语义程序属性的泛化能力有限。我们还展示了模式定义的两个示例应用。为评估 code2vec 和 code2seq 的鲁棒性，我们的方法采用更小的扰动，并且相比以往方法只需进行少得多的尝试即可找到对抗样本。在提升 code2vec 和 code2seq 的准确率方面，我们基于对抗训练设计的新技术使 code2seq 在 Java-med 和 Java-large 上取得了最先进的结果。

- 我们计划进一步完善并部署我们的实现，以便向用户提供更细粒度的预测标签解释，例如，方法中的哪个表达式导致了输出中的某个特定词，从而帮助用户更好地理解模型做出预测时所采用的推理。
- 我们计划进一步诊断代码摘要模型，以设计一个通用、系统的框架，提升代码摘要模型的准确率和鲁棒性。
- 我们计划将这项技术用于研究其他代码模型为解决编程语言中的不同任务而学习到的模式。

参考文献

Uri Alon、Omer Levy 和 Eran Yahav。2019a。code2seq：从代码的结构化表示生成序列。国际学习表征会议 (2019)。Uri Alon、Meital Zilberstein、Omer Levy 和 Eran Yahav。2019b。Code2Vec：学习代码的分布式表示。*ACM* 程序设计语言会议录 POPL (2019)。Dzmitry Bahdanau、Kyunghyun Cho 和 Yoshua Bengio。2015。通过联合学习对齐与翻译实现神经机器翻译。国际学习表征会议 (2015)。David Bau、Jun-Yan Zhu、Hendrik Strobelt、Bolei Zhou、Joshua B. Tenenbaum、William T. Freeman 和 Antonio Torralba。2019。生成对抗网络的可视化与理解。国际学习表征会议 (2019)。Alan W Biermann 和 JA Feldman。1972。文法推断研究成果综述。载于模式识别前沿。爱思唯尔，31–54。N. Carlini 和 D. Wagner。2017。迈向神经网络鲁棒性的评估。载于*2017*年*IEEE*安全与隐私研讨会（*SP*）。IEEE计算机学会。Hong Chen、Zi Jian Xu、Zi Qiang Liu 和 Song Chun Zhu。2006。用于布料建模与素描的复合模板。载于*2006*年*IEEE*计算机学会计算机视觉与模式识别会议（*CVPR'06*），第1卷。IEEE，943–950。Kyunghyun Cho、Bart van Merriënboer、Caglar Gulcehre、Dzmitry Bahdanau、Fethi Bougares、Holger Schwenk 和 Yoshua Bengio。2014。使用循环神经网络编码器–解码器学习用于统计机器翻译的短语表示。载于

*2014*年自然语言处理实证方法会议论文集（*EMNLP*）。Alexis Conneau 和 Guillaume Lample。2019。跨语言语言模型预训练。载于神经信息处理系统进展第*32*卷。7059–7069。Jia Deng、Wei Dong、Richard Socher、Li-Jia Li、Kai Li 和 Li Fei-Fei。2009。ImageNet：大型分层图像数据库。载于*2009*年*IEEE*计算机视觉与模式识别会议。IEEE，248–255。Jacob Devlin、Ming-Wei Chang、Kenton Lee 和 Kristina Toutanova。2019。BERT：用于语言理解的深度双向变换器预训练。载于*2019*年计算语言学协会北美分会会议论文集：人类语言技术，第*1*卷（长短篇论文）。Jacob Devlin、Rabih Zbib、Zhongqiang Huang、Thomas Lamar、Richard Schwartz 和 John Makhoul。2014。用于统计机器翻译的快速、鲁棒神经网络联合模型。载于计算语言学协会第*52*届年会论文集（第*1*卷：长篇论文）。Patrick Fernandes、Miltiadis Allamanis 和 Marc Brockschmidt。2019。结构化神经摘要。国际学习表征会议 (2019)。Timon Gehr、Matthew Mirman、Dana Drachsler-Cohen、Petar Tsankov、Swarat Chaudhuri 和 Martin Vechev。2018。Ai2：利用抽象解释对神经网络进行安全性与鲁棒性认证。载于*2018*年*IEEE*安全与隐私研讨会（*SP*）。IEEE，3–18。Ian J. Goodfellow、Jonathon Shlens 和 Christian Szegedy。2015。解释并利用对抗样本。国际学习表征会议 (2015)。Feng Han 和 Song-Chun Zhu。2008。利用属性文法自底向上/自顶向下进行图像解析。*IEEE*模式分析与机器智能汇刊 (2008)。Alex Krizhevsky、Ilya Sutskever 和 Geoffrey E Hinton。2012。利用深度卷积神经网络进行ImageNet分类。载于神经信息处理系统进展。1097–1105。José Oncina 和 Pedro Garcia。1992。在多项式更新时间内推断正则语言。载于模式识别与图像分析：第四届西班牙研讨会精选论文。49–61。Gordon Plotkin 和 Gilles Kahn。1987。操作语义的结构化方法。载于*STACS'87*：第四届年度研讨会论文集，主题为。奥胡斯大学，22–39。Aditi Raghunathan、Jacob Steinhardt 和 Percy Liang。2018。针对对抗样本的认证防御。国际学习表征会议 (2018)。David Silver、Aja Huang、Chris J. Maddison、Arthur Guez、Laurent Sifre、George van den Driessche、Julian Schrittwieser、Ioannis Antonoglou、Veda Panneershelvam、Marc Lanctot、Sander Dieleman、Dominik Grewe、John Nham、Nal Kalch- brenner、Ilya Sutskever、Timothy Lillicrap、Madeleine Leach、Koray Kavukcuoglu、Thore Graepel 和 Demis Hassabis。2016。通过深度神经网络和树搜索掌握围棋。《自然》(2016年1月1日)。David Silver、Julian Schrittwieser、Karen Simonyan、Ioannis Antonoglou、Aja Huang、Arthur Guez、Thomas Hubert、Lucas Baker、Matthew Lai、Adrian Bolton、Yutian Chen、Timothy Lillicrap、Fan Hui、Laurent Sifre、George van den Driessche、Thore Graepel 和 Demis Hassabis。2017。无需人类知识掌握围棋。《自然》(2017年10月1日)。Andrew Stevenson 和 James R Cordy。2014。软件工程中文法推断研究综述。《计算机编程科学》96 (2014), 444–459。Christian Szegedy、Wojciech Zaremba、Ilya Sutskever、Joan Bruna、Dumitru Erhan、Ian Goodfellow 和 Rob Fergus。2013。神经网络的有趣特性。国际学习表征会议 (2013)。Hugo Touvron、Andrea Vedaldi、Matthijs Douze 和 Hervé Jégou。2019。修正训练与测试分辨率差异。载于神经信息处理系统进展。8250–8260。Ashish Vaswani、Noam Shazeer、Niki Parmar、Jakob Uszkoreit、Llion Jones、Aidan N Gomez、Łukasz Kaiser 和 Illia Polosukhin。2017。注意力机制就是你所需要的一切。载于神经信息处理系统进展。5998–6008。Oriol Vinyals、Igor Babuschkin、Wojciech M. Czarnecki、Michaël Mathieu、Andrew Dudzik、Junyoung Chung、David H. Choi、Richard Powell、Timo Ewalds、Petko Georgiev、Junhyuk Oh、Dan Horgan、Manuel Kroiss、Ivo Danihelka、Aja Huang、Laurent Sifre、Trevor Cai、John P. Agapiou、Max Jaderberg、Alexander S. Vezhnevets、Rémi Leblond、Tobias Pohlen、Valentin Dalibard、David Budden、Yury Sulsky、James Molloy、Tom L. Paine、Caglar Gulcehre、Ziyu Wang、Tobias Pfaff、Yuhuai Wu、Roman Ring、Dani Yogatama、Dario Wünsch、Katrina McKinney、Oliver Smith、Tom Schaul、Timothy Lillicrap、Koray Kavukcuoglu、Demis Hassabis、Chris Apps 和 David Silver。2019。利用多智能体强化学习达到《星际争霸II》宗师级水平。《自然》(2019年11月1日)。Ke Wang 和 Mihai Christodorescu。2019。COSET：用于评估神经程序嵌入的基准。*arXiv* 预印本 *arXiv:1905.11445* (2019)。Ke Wang 和 Zhendong Su。2020。融合式、精准的语义程序嵌入。载于第*41*届*ACM SIGPLAN*国际编程语言设计与实现会议论文集（*PLDI '20*）。Shiqi Wang、Kexin Pei、Justin Whitehouse、Junfeng Yang 和 Suman Jana。2018。利用符号区间对神经网络进行形式化安全分析。载于第*27*届*USENIX*安全研讨会论文集（*SEC'18*）。1599–1614。

Matthew D. Zeiler 和 Rob Fergus。2014年。《可视化和理解卷积网络》。收录于 计算机视觉 – *ECCV 2014*。818–833。Matthew D Zeiler、Graham W Taylor 和 Rob Fergus。2011年。用于中高层特征学习的自适应反卷积网络。收录于*2011*年国际计算机视觉会议。IEEE，2018–2025。