北京大学

博士研究生学位论文

题目：感知关键约束的深度学习程序生成技术研究

姓　　名：孙泽宇

学　　号：1901111277

院　　系：计算机学院

专　　业：计算机软件与理论

研究方向：高可信软件理论与技术

导　　师：张路 教授

二〇二二 年 六 月

版权声明

摘要

程序的编写是软件开发中的主要活动。提高程序编写的效率一直是软件工程研究关注的重要问题。长期以来，许多研究人员致力于通过程序生成、程序搜索和程序补全等方法来提升程序编写的效率。

其中，基于自然语言描述的程序生成被认为是提高程序编写效率的重要途径，并且已经引起了学术界和工业界的广泛关注。程序生成方法可以根据用户需求来自动生成软件源代码。准确的程序生成方法有望减轻了软件开发人员的负担，让他们可以更加关注程序设计本身。

基于深度学习的程序生成方法借鉴了自然语言处理中的基于深度神经网络的机器翻译方法，试图将输入的自然语言描述自动转换为对应程序。然而，不同于机器翻译的是，程序生成方法所生成的程序与机器翻译技术所得到的自然语言文本有着根本的不同。相较于自然语言文本，程序中含有更多且更严格的约束，其中包括：1）程序语法约束，生成的程序必须满足程序的语法性质，如果生成的程序不满足语法性质，那其一定无法通过编译，即，其一定是一段错误程序；2）语义约束，生成的程序中变量、函数名必须满足程序规定的性质。如果变量、函数名的命名不符合预定义的语义规定，那么其可能会导致运行错误，即，其一定是一段语义不正确的程序；3）任务特定的约束，在程序生成任务中，生成的程序与对应的自然语言描述之间应该满足扰动约束。对自然语言进行不影响程序算法的词语替换之后，程序的结构应该保持不变，否则，该程序可能不正确。

使得基于深度学习的程序生成方法感知到以上三种约束有望使得现有程序生成的效果得到提升。然而，将这些约束引入基于深度学习的程序生成方法中是一个非常具有挑战性的问题。现有的工作虽然分别从程序语法约束感知和语义约束感知的角度进行了研究，然而，其在程序生成中的效果仍然不够理想。

本文针对现有基于深度学习的程序生成方法中的程序语法约束、语义约束和任务特定的约束的感知问题在现有工作的基础上展开进一步的研究。

对于程序语法约束的感知，本文关注于抽象语法树知识，提出了一种基于 Transformer 结构的神经网络技术（TreeGen）。该技术对程序语法中结构与语法规则进行编码，提升了神经网络对程序语法的感知能力。本文在 Python 上的标准数据集 HearthStone 和两个 Lambda 演算上的标准数据集 ATIS 和 GEO 上评估了所提出模型的效果，TreeGen 在 HearthStone 上的表现比之前的最好的方法高出 9 个百分点，并且在 ATIS (89.1%) 和 GEO (89.6%) 数据集上取得了基于神经网络的方法的最佳生成准确率。

对于语义约束的感知，本文关注于变量名、函数名命名约束，提出了一种适合于神经网络的感知变量名、函数名命名约束的迭代筛选算法。该算法可以与现有的神经网络技术相结合，以辅助神经网络训练的方式提升现有的神经网络对于变量、函数名命名的感知能力。本文将迭代筛选算法与 TreeGen 相结合。在 Github 上所爬取的数据集上的实验结果表明，本文所提出方法相比于不使用该方法的技术生成程序的准确率提高了 8 个百分点。

对于任务特定约束的感知，本文关注于一种任务特定的扰动约束，提出了一种适合于神经网络的感知扰动约束的检测和修复算法。该算法先通过检测的方法来检测程序生成模型生成的程序是否满足对应扰动约束，并报出生成程序中不满足扰动约束的问题。在报出问题的基础上，本文提出了一种基于集成学习机制的自动提升方法来修复所报出的问题，进而提升程序生成技术的效果。实验结果表明，本文所提出的检测和修复算法在常用的程序生成工具上发现并成功修复了 3%-8% 的生成不正确的程序。

此外，本文所研究约束感知问题并不完全局限于程序生成，在通用的无属性图分类任务及机器翻译任务中也存在类似的约束感知问题。本文进一步将所研究方法推广至这两个任务之上。本文将所提出的语义约束感知方法推广到通用图神经网络中的无属性图分类任务之上。所用方法在布尔可满足性问题求解（SAT）及最大独立集求解任务上进行验证，其预测错误率相比于各自任务对应的现有最佳方法下降了 76% 和 39%。本文将基于集成学习的扰动约束感知方法推广到机器翻译之上。所用方法在谷歌翻译及 Transformer 翻译器上进行验证，其检测技术平均为其检测出 40% 的翻译问题，而其修复技术平均为谷歌翻译和 Transformer 模型修复了 53% 和 52% 的问题。

综上，本文提出了一系列技术方法，形成一套程序生成技术体系。该技术体系在一系列程序生成的数据集上超过了最优方法，部分数据集上最优记录已经保持 3 年。同时，这些技术也被用到多个其他领域，除了上面介绍的布尔可满足性求解、最大独立集求解、机器翻译测试等问题外，还被同行用于代码搜索、程序修复等领域，所有应用都取得了目前最优的效果。

关键词：程序生成，神经网络，约束感知

Research on Key-Constraints-Aware Program Generation Based on Deep Learning

Zeyu Sun (Computer Software and Theory)
Supervised by Prof. Lu Zhang

ABSTRACT

Programming is important in software development. In order to improve the efficiency of programming, many researchers are devoted to program generation, code search, and code completion.

As a representative task in these areas, program generation has attracted extensive attention from both academia and industry.Given an input natural language description, a program generation system generates the target program automatically.

Existing approaches apply deep neural-based machine translation technology to convert an input natural language description into the target program. However, the program generated by program generation is different from the natural language generated by machine translation. The program contains a lot of constraints that are not included in natural language: 1) The grammar constraint. The generated program must satisfy the predefined grammar, otherwise, its compilation should fail; 2) The semantics constraint. The variable names and function names in the generated program must satisfy a property that when they are renamed, the semantics should not be changed, otherwise their semantics must be incorrect; 3) The task-specific constraint. In the program generation task, the generated program and the input natural language description should satisfy the perturbation constraint. If the natural language description is perturbed without hurting the semantics, the structure of the generated program should be unchanged, otherwise, the program may be incorrect.

Encoding the constraints into program generation may improve the performance. However, it is a very challenging problem. Although the existing work has focused on grammar encoding and semantics encoding, their performance is still not ideal.

To tackle this problem, this thesis still focuses on encoding the grammar constraint, the semantics constraint, and the task-specific constraint in program generation.

For the grammar constraint, this thesis proposes a tree-based transformer model, TreeGen.

TreeGen introduces a novel AST reader to incorporate grammar rules and AST structures into the network. The evaluation was conducted on a Python benchmark, HearthStone, and two semantic parsing benchmarks, ATIS and GEO. TreeGen outperformed the previous state-of-the-art approach by 4.5 percentage points on HearthStone, and achieved the best accuracy among neural network-based approaches on ATIS (89.1%) and GEO (89.6%).

For the semantics constraint, this thesis proposes preferential labeling. Preferential labeling satisfies the property of variable names and function names asymptotically. The experiments were conducted on a dataset crawled from Github. The results show that the proposed preferential labeling improves the accuracy of TreeGen by 8 points.

For the task-specific constraint, this thesis represents a kind of task-specific constraint as a metamorphic constraint. Based on this metamorphic constraint, this thesis proposes a detection approach, CAT, to automatically detects the corresponding issues in a program generation system. Then, CAT uses an ensemble learning-based approach to fix the detected issues. The proposed CAT was evaluated on a widely used program generation tool. The experimental results show that CAT can detect and repair 3%-8% incorrectly generated program.

Furthermore, the problem of encoding constraints not only exists in program generation but also exists in the field of unattributed node classification and machine translation. Thus, this thesis generalizes the proposed approaches to these two fields. This thesis generalizes preferential labeling to the unattributed node classification task. Experimental results show that preferential labeling successfully improves the performance of unattributed node classification, where the number of errors drops by 39% in the MIS problem and 76% in the SAT problem. Meanwhile, this thesis generalizes CAT to machine translation. The experimental results show that CAT improves the performance of machine translation. CAT fixes 53% and 52% of the issues for Google Translate and Transformer.

To sum up, this thesis proposes a program generation system. The proposed program generation system outperforms the state-of-the-art approaches on several program generation datasets. Furthermore, TreeGen keeps the best performance on some datasets in the past 3 years. At the same time, the proposed system is also used in many other fields including the above-mentioned SAT solving, MIS solving, and machine translation testing. The system is also used by peers in code search, program repair, and other fields, and all these applications have achieved the best results so far.

KEY WORDS: Code Generation, Neural Network, Constraints-Aware

目录

第一章　引言

1.1　问题的提出

随着人类社会的发展，信息化逐渐成为了社会发展的关键词之一，软件也逐渐成为了信息化服务设施的基本构建元素。在信息化的过程中，软件所涉及的功能越来越复杂，其程序对应的代码规模也日益增长。而软件开发主要依赖于开发者的人工劳动付出，如何有效地提高开发者的开发效率、减轻开发者的开发负担也成为了软件工程中相关研究领域所关注的问题。

长期以来，许多研究人员通过研究辅助软件开发的方法来提高软件开发的自动化水平。其中具有代表性的技术便是程序生成方法。给定一段自然语言描述，程序生成方法可以帮助使用者自动生成该描述所对应的目标代码。举例来说，给定一段自然语言“打开 A 文件”，程序生成方法可以生成一段指定语言的代码“f = open(A)”（Python）。

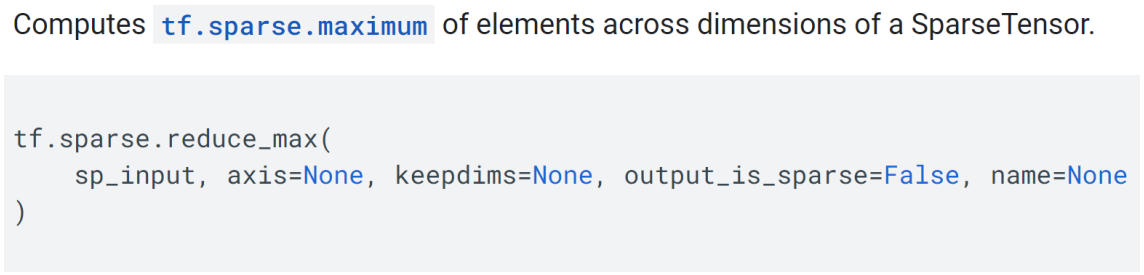

图 1.1　来源于 TensorFlow 的 API 文档的 API 描述及其代码

程序生成方法可以被广泛应用于开发者的开发过程以及特定程序语言初学者的学习过程等场景之中。具体来说，开发者常常需要以查阅 API 说明文档的方式辅助其完成软件开发（例如，查询如何计算 SparseTensor 这类 Tensor 中某一维度的最大值，如图 1.1）。该查阅的过程极大地消耗了开发者的开发时间，若使用程序生成方法，开发者只需要将自然语言描述的需求作为输入，便可以直接获得其所需要的代码，从而帮助开发者缩短开发时间。对于程序语言的初学者而言，其通常会存在不知道如何使用特定程序语言去实现特定功能的问题。例如，Python 初学者使用 Python 语言对 A 和 B 两个矩阵做矩阵乘法（“numpy.matmul(A, B)”）。若使用程序生成方法，初学者只需要将一段自然语言描述“对 A 和 B 两个矩阵做矩阵乘法”作为输入，程序生成方法便可自动生成其对应的代码，以帮助程序语言初学者完成学习的目的。

程序生成作为提高软件开发自动化水平的重要途径，已经引起了学术界和工业界的广泛关注。程序生成利用一些技术来自动生成软件源代码，从而达到根据用户需求

进行自动化编程的目的。高质量的程序生成有望大大减轻程序员的开发负担，使程序员可以更加关注程序的设计。即便程序生成方法没有完整地生成目标程序，其也会生成参考程序以辅助开发者完成开发工作。

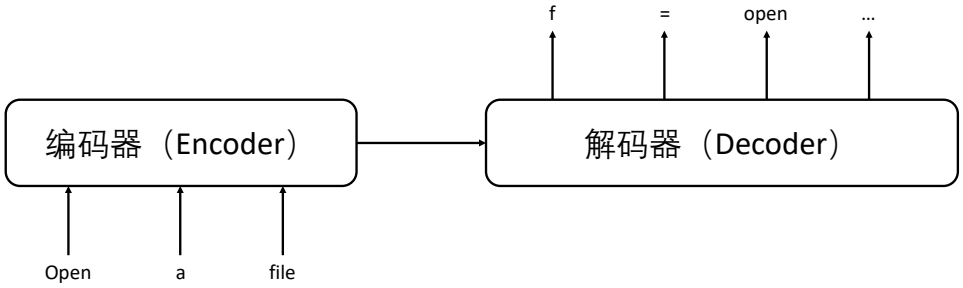

图 1.2　Sequence-to-sequence 模型

深度学习技术在被提出后，于近十几年内蓬勃发展，使得人工智能技术有了革命性的突破。在大量训练数据和高性能计算资源的支持下，深度学习技术已在图像识别[1]、语音处理[2]、自然语言处理[3,4]、和软件工程[5]等领域中取得了令人瞩目的效果。

Ling et al.（2016）[6]首次将自然语言翻译中的基于序列到序列（sequence-to-sequence）的深度学习框架与程序生成相结合。该工作把程序当作一种新的自然语言并基于程序分词后所得程序词序列来生成代码。该工作的大致框架如图 1.2所示，给定一段自然语言描述，一个用数据训练好的神经网络模型可以自动生成对应的程序。

该方法有一定的合理性，却忽视了程序语言和自然语言的区别，直接将程序语言当作自然语言来完成生成任务。与自然语言不同的是，程序中包含了众多的约束（程序生成系统应该具有的性质）。如果直接忽视这些约束，会使得程序生成系统生成的程序可能不具有该有的性质，进而导致错误的发生。如何在程序生成任务中感知这些约束也是后续工作的研究重点。

1.2　研究现状

为了更好地感知约束并提升程序生成方法效果，本节对相关的研究现状进行调研。在程序生成中，已有一些方法使用了约束感知的技术。为此，本节先对程序生成中约束感知的方法进行介绍（第1.2.1节）。同时，在代码相关领域的技术中也存在编码程序的问题，也需要感知程序当中的约束。为了了解该方面的研究现状，本节进而对代码相关任务中约束感知的方法进行介绍（第1.2.2节）。最后，许多特定机器学习任务中往往存在任务特定的约束感知问题。其问题与本文所研究的约束感知问题具有一定程度上的相似性。因此，本节也对通用领域下的机器学习任务中约束感知的方法进行介绍（第1.2.3节）。

1.2.1 程序生成任务中的约束感知

程序生成任务中的约束指在程序生成领域下把生成系统看作一个函数，该函数必须具有的性质。

相比于自然语言，程序对于语法约束的要求更为严格。如果一个程序生成系统生成的程序不满足其对应的语法约束条件，那么该程序一定无法通过编译。换句话说，该程序一定是一个错误程序。研究者们意识到了这个问题，并且为程序语法约束感知问题开展了大量研究。本章将在第1.2.1.1节对这类工作进行介绍。

除此之外，也有一部分研究者将注意力放到程序语义之上。语义定义为从程序到状态之间的函数的映射。对该映射进行约束的逻辑表达式就是语义约束。现有关于语义约束的研究主要关注于程序中调用关系编码和变量、函数名命名关系的编码。本章将在第1.2.1.2节对这类工作进行介绍。

特别的，本文主要关注于基于深度学习的程序生成中约束感知方法。因而重点调研了该领域及其相关领域。

1.2.1.1 语法约束的感知

语法约束是程序语言中必不可少的一部分，研究者们致力于将程序中的语法约束编码到神经网络当中以实现语法约束的感知。

抽象语法树（Abstract Syntax Tree，AST）是程序中最具代表性的语法信息。现有的主流工作在抽象语法树的基础上实现程序语法约束的感知。该类技术的整体思想是先将抽象语法树通过遍历的方式转换成一段序列，然后利用基于序列的模型对其建模以实现语法约束感知。以下将对该类技术进行详细介绍。

为了编码抽象语法树，一些研究者利用传统机器学习的方法对抽象语法树进行建模。Raychev et al.（2016）[7]和 Bielik et al.（2017）[8] 分别利用上下文无关文法和决策树对抽象语法树建模并借此预测代码。

深度学习方法相比传统机器学习方法而言，其结构更为复杂。该类方法需要根据抽象语法树的结构定制新的网络。Rabinovich et al.（2017）[9] 提出了一种抽象语法网络（Abstract Syntax Networks，ASNs），该网络是基于循环神经网络的序列到序列的框架的扩展形式。与 Ling et al.（2016）[6]所提出的序列模型不同，其解码器是由许多子模块组成，每个子模块都与抽象语法树的一个语法结构相对应。在解码过程中，循环神经网络的状态从一个模块传递到另一个模块，借此实现模块之间信息的传递。在抽象语法树的生成过程中，ASNs 根据当前的状态选择不同的子模块执行不同的网络，进而实现树的生成。

与之不同的是，Dong 和 Lapata（2016）[10]以及 Dong 和 Lapata（2018）[11]提出了一种

基于层级的树解码器 SEQ2TREE。解码器根据已生成程序的树结构得到的编码结果之后，利用循环神经网络生成程序词序列。当所生成的树节点为程序中的非终结节点（包括程序中的括号等词汇）时，该技术将非终结节点的状态输入到预测树编码器的下一层，直到没有非终结节点为止。然而，该工作还是以生成程序的词序列为目标而非生成语法规则序列，这会导致生成的程序可能不符合语法定义。

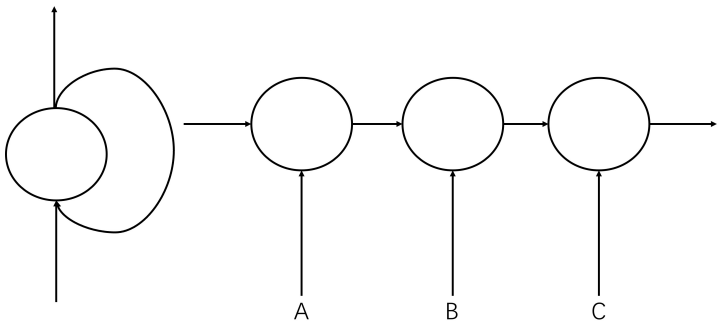

图 1.3　循环神经网络

Yin 和 Neubig（2017）[12]将语法规则序列引入到程序生成模型当中，并使用了基于 LSTM 的序列到序列的模型对输入自然语言描述及其输出的部分语法规则序列进行编码。同时，该方法也按照预定义语法规则预测并生成程序。以这种方式生成的程序都是符合语法规则的程序。并且，他们还使用了一种 parent feeding 的技术。该技术在预测过程将解码器中每个结点的中间表示与其父结点的表示相结合，一定程度上实现了语法规则结构的编码。在此基础上，Yin et al.（2018）[13]再次提出使用一种结构化的自编码器技术丰富程序数据集的多样性，提升了神经网络生成程序的能力。基于这两个工作，Yin 和 Neubig（2018）[14]提出了一个感知语法约束的程序生成工具。在该工具的基础上，部分研究者对其进行语法结构上编码的修改提出了一系列工作[15,16]。然而，上述方法都基于循环神经网络（如图 1.3所示，其通过自回归运算来完成循环的过程），其受制于循环神经网络的“自回归性”（循环神经网络的结点主要依赖于其生成顺序之前的结点表示）及长程依赖问题。其方法仍有较大的提升空间。

Jiang et al.（2021）[17] 在感知程序语法约束的基础上，探究了生成程序的搜索步骤对结果的影响，并给出了一种选择合适搜索步骤的方法。Xiong 和 Wang（2022）[18] 提出了一种通用的感知语法约束的程序生成框架，该框架可以将程序生成任务通过一定方式转换为一系列语法规则分类任务以生成程序。

1.2.1.2　语义约束的感知

除语法约束外，语义约束也是程序语言中必不可少的一部分。现有的感知语义约束的工作主要关注于语义约束中 API 调用图等调用约束的感知问题或者是变量、函数

名等标识符命名约束的感知问题。

调用约束的感知 调用关系包含了程序中部分可编译性的约束，如果程序调用关系错误（类型错误、参数数量错误等）则其一定无法通过编译。现有的调用关系（API 调用图等图关系）编码工作主要关注于不同图类信息表示方法。Nguyen 和 Nguyen（2015）[19]对 API 的调用关系进行图建模表示。图中的节点包括动作（即 API 调用、重载操作和域的访问）和控制点（即控制结构的分支 if、while、for 等）。图上的边则表达了这些节点的依赖关系。Nguyen 和 Nguyen（2016）[20] 以及 Gu et al.（2017）[21]提出通过序列来表示 Java 方法中的 API 关系。Allamanis et al.（2018）[22] 提出了一种新的数据流图建模方式来表示语义关系（该数据流图包括使用关系和写入关系），并使用基于图的神经网络来表示程序中其所定义的语法和语义特征。Lyu et al.（2021）[23] 提出了一种训练 API 依赖关系的方法，为 API 中所含有的程序语义单独设计了一个 API 依赖图建模模块以提升神经网络对 API 语义的感知能力。

命名约束的感知 对于语义约束中变量、函数等标识符命名约束来说，其具体形式可以表示为命名名称不会影响或改变程序语义的性质。其可以表示为，for all a, b in Prog, rename(a, b) -> [a]=[b]，其中 Prog 是所有程序的集合，rename(a, b) 表示 a 和 b 之间仅存在变量改名的不同，[] 是语义映射。现有的主流工作主要分为两类。

1）直接使用原始变量、函数名进行编码的方法[12]。该类方法直接使用所获取的训练数据中不加任何修改的变量、函数名进行编码并训练。

2）将变量、函数名替代为预定义的变量、函数名的方法[24,25]。该类方法将每条数据中出现的变量、函数名按照一定顺序替换为预定义的名称。举例来说，对于一条赋值语句“a = b + c”，第一个出现的变量名“a”会被替换为变量名“变量 1”，而第二/三个出现的变量名“b”/“c”会被替换为变量名“变量 2”/“变量 3”，即最后会将该赋值语句转换为“变量 1 = 变量 2 + 变量 3”。

1.2.2 基于深度学习的程序分析任务中的约束感知

和程序生成相近，基于深度学习的程序分析任务中的约束感知研究也关注程序生成任务中所关注两种约束：程序语法约束和语义约束。

1.2.2.1 语法约束的感知

现有的程序分析任务中的语法约束感知研究同样是依赖于抽象语法树来表示程序中的语法约束。在该类研究中，抽象语法树的编码方式分为两种：1）基于路径的技术，2）基于树的技术。

基于路径的技术 Alon et al.（2019）[26]提出了一种 Code2Vec 方法，该方法目的是利用抽象语法树和神经网络技术将程序转换成对应表示的数值向量。特别的，该方法提出了一种基于路径的技术，将抽象语法树转换为一系列语法树上的路径。该方法通过编码这些语法树路径实现其语法约束的编码。与之同样的，Alon et al.（2018）[27] 利用此技术提出了一系列应用方法[27,28]。但是该方法存在路径爆炸的问题，其难以在程序生成中直接应用。

基于树的技术 与上述方法不同的，其他的抽象语法树编码方法倾向于使用树或者图结构对程序语法进行编码感知[25,29]。Mou et al.（2016）[30]首次在程序中提出了一种树卷积神经网络的思想，将程序的抽象语法树用一种树结构的网络进行编码。

Li et al.（2018）[31]和 Ben-Nun et al.（2018）[32]通过对程序抽象语法树进行遍历将其转化为一段序列，并进一步根据这段序列编码程序。Fernandes et al.（2018）[33]、Wei et al.（2019）[34]以及 Zhang et al.（2019）[35]使用了类似于程序生成中抽象语法树编码的思想，将抽象语法树转成图。其进一步借用图神经网络技术对其进行编码，从而提升神经网络对语法约束的感知能力。

Jiang et al.（2021）[36] 将程序抽象语法树和一种基于预训练模型的程序编码技术进行结合[37]，利用大规模数据对程序进行预训练以提升神经网络对程序的编码能力。

曹英魁 et al.（2021）[38]使用了抽象语法树对于代码转换进行编码，提升了神经网络在代码转换领域感知语法约束的能力。

1.2.2.2 语义约束的感知

主流的技术专注于对语义约束中的调用关系，Sui et al.（2020）[39]使用了图结构对程序中的数据流图等调用关系进行表示。同时，其使用图神经网络对该调用关系进行编码，从而使得神经网络可以编码程序调用关系约束，帮助神经网络感知程序语义。

Zhang et al.（2021）[40]将程序中的依赖图通过随机游走的方式转为表示图结构的多串序列，并使用 Transformer 模型对这些序列进行编码，从而实现感知语义约束的目的。

1.2.3 机器学习其它应用中的约束感知

机器学习其他应用中往往也存在约束感知的问题。其具体表现为：在任务特定领域下的约束感知问题。举例来说，在动物识别任务中，约束通常是对输入图片进行镜像对称变换不会影响输出结果。在机器翻译领域中，约束通常是对输入的自然语言进行语义相近的变换其输出的翻译不会发生较大改变。在布尔可满足性问题求解 (SAT) 领域中，约束通常是输入公式的变量名称发生改变，其输出结果不应该发生改变。

在这些约束感知的过程中，基于任务的不同往往会使用不同的约束感知方法。这类方法通常分为：1）基于数据增强的约束感知，2）基于集成学习的约束感知，3）其他约束感知方法。

1.2.3.1 基于数据增强的约束感知

为了感知任务特定的约束，许多研究者致力于通过数据增强的方法利用约束来扩充数据集。用扩充过的数据集进行训练可以使得机器学习模型学习到对应的约束。本节对图像处理和自然语言处理领域的数据增强方法进行简单介绍。

在图像处理领域，数据增强方法通常是对输入图片进行一系列等价变换。Wu et al.（2015）[41]以及 Mikoajczyk 和 Grochowski（2018）[42]对图片颜色进行修改实现数据集的增强以完成图像识别任务。Zhong et al.（2020）[43]则是对图片的一部分进行随机移除。除此之外，也有方法对图片进行旋转、折叠实现数据增强[44]实现对应任务特定的约束感知。

在自然语言处理领域，数据增强方法通常是对输入的自然语言进行语义相近的变换。Zhang et al.（2015）[45]采用了近义词替换的方法完成新闻和评论分类任务。类似的，Wei 和 Zou（2019）[46]随机对输入中的词语进行交换或删除来实现数据增强以完成自然语言分类任务。Feng et al.（2019）[47]则提出了一种保持整体语义不变的语句修改方法。

1.2.3.2 基于集成学习的约束感知

也有些研究者使用集成学习的方法感知任务特定的约束。集成学习方法可以从数据的角度对相似数据的输入输出结果进行集成，也可以从模型的角度对相似模型的输出进行集成。

从数据的角度而言，现有很多研究者致力于对输入进行一定不影响输出结果的变化，进而产生多个输入和其对应的输出，最后将不同的输出进行结合以实现集成。Nesti et al.（2021）[48]对输入图片进行少量修改生成新的输入数据，最后将不同输入数据的输出进行集成。Dong et al.（2021）[49]则对输入文本进行修改，并以与前者同样的方式进行集成。

从模型的角度而言，现有很多研究者致力于使用多个相近模型对同一个数据进行预测不同的输出进行集成。Wang et al.（2014）[50]对于同一个自然语言输入使用多个贝叶斯模型对其计算并通过投票的方式对结果进行分类预测。与之相似的，Chalothom 和 Ellman（2015）[51]则是使用了 SVM 模型、贝叶斯模型等多个不同的模型进行集成。

1.2.3.3 其他约束感知方法

为了感知任务特定的约束，研究者往往采用直接对输出添加约束的方式，对不符合任务特定对应约束的输出进行修改或者删除操作。

Li et al.（2018）[52] 在解决最大独立集问题的时候，对机器学习模型输出的结果进行后处理。该后处理使用了一种搜索策略，使得其搜索到的结果一定是一个独立集。与之类似的，Zhang et al.（2020）[53]在求解布尔表达式问题时，将神经网络模型的输出结果输入到 LocalSAT 求解器中，从而保证其输出结果的正确性。

1.2.4 尚待解决的问题

本章对已有相关工作进行了调研。基于调研的内容，程序生成任务的约束主要可以分为：1）程序语法约束、2）语义约束和 3）任务特定的约束三类（程序生成任务还存在有其他输入自然语言相关的约束。本文主要关注于程序部分的约束）。虽然现有许多工作专注于感知约束的程序生成，但是仍然存在如下尚待解决的问题。

1.2.4.1 程序语法约束的感知

如上文所述，现有部分工作已经意识到感知程序语法约束的重要性，并尝试采用抽象语法树的方式将语法约束编码到神经网络之中[12,54-56]。

然而，之前的工作都是基于循环神经网络。该类工作难以很好地感知程序语法约束，因为循环神经网络是一种用以表示一串序列的“扁平”化神经网络。举例来说，在抽象语法树中，一个子节点和其父节点是相邻节点的关系，而如果利用“扁平”化的方式将其转换为一段序列（通常是先序遍历序列），其往往在这段序列中难以保持原本应有相邻节点的关系。该问题会导致子节点和父节点之间丧失了“父-子”关系，从而难以感知到抽象语法树本身含有的语法树结构的信息。

同时，程序中也存在大量语法长程依赖关系，其中，代码中的一个元素可能依赖于另一个存在于多行之前的元素。例如，第 100 行的一条变量赋值语句 “Max_Length = 100” 可能依赖于第 50 行的一条 If 语句 “if len(a) < Max_Length:’’。依赖循环神经网络的现有技术会受制于神经网络本身所含有的长程依赖问题的影响，从而难以解决该类依赖问题[57]。

以上问题限制了现有的神经网络技术感知语法约束的能力，从而限制了程序生成方法的发展。因此，如何解决以上问题是程序语法约束感知的关键。

1.2.4.2 语义约束的感知

对于语义约束的感知，本文关注于变量、函数名的命名约束。该约束可以表示为：标识符可以在不改变程序语义的情况下任意重命名[24,25]。现有的两类关于命名约束感知的工作存在难以感知该性质的问题，进而造成了一定的编码缺陷：

- 直接使用原始变量、函数名进行命名约束感知的方法对应问题[12]：该类工作直接忽视了变量、函数名所拥有的性质，将其当作自然语言中的词汇来进行处理。这

会使得神经网络对变量、函数名的命名风格的改变十分敏感，其会十分依赖于训练集的变量、函数名的命名质量。而在常用的 Github 开源数据中，其变量、函数名命名方式往往在不同的项目中有着不同的风格。这种不同的命名风格会使得其难以在不同项目上的训练样本之间共享所学到的约束。同时，神经网络也难以将学到的约束推广到具有不同命名风格的新项目的样本之上。与之相同的，该问题也存在于变量、函数名命名质量较低的程序数据上。

- 将变量、函数名替代为预定义的变量、函数名的方法对应问题[24,25]：这类方法无法从根本上解决变量、函数名编码的问题，而是使用旁路的方法。这会导致一些变量、函数名编码本身携带命名语义约束的丢失。因此，用该方法生成的程序难以让开发者理解。

上述问题制约着语义约束感知的发展，一个合理的变量、函数名编码方法应避免上述缺陷并满足相应的性质。

1.2.4.3 任务特定的约束感知

如同机器学习任务中的任务特定的约束感知一样，程序生成中也有特定于自身的约束。该任务特定的一种约束可以表示为扰动约束：对自然语言进行不影响程序算法的词语替换之后，程序的结构应该保持不变。举例来说，对于一个输入“计算 A+3”，如果将其修改为“计算 A+5”，那么其对应的程序应该从“A+3”也变成“A+5”现有的程序生成工作忽视了任务特定的约束感知问题。

由于现有工作并不刻意感知输入自然语言描述和输出程序代码之间的关联关系，这导致现有程序生成方法在很多情况下具有较差的生成效果。如表 1.1所示，程序生成方法（CodeGPT）在生成自然语言“returns the first child element of a node which matches the given tag name.”的输出时，其得到 Java 代码“T function (T arg0, String arg1) { for (T loc0: children()) { if (arg0.getTagName().equals(arg1)) { return loc0;}} return null;}”。然而，当我们把输入中的“which”修改为具有同样含义的“that”时，其输出的代码发生了巨大改变，即生成了代码“T function (Document arg0, String arg1) { return (T) arg0.getFirstChild(arg1);}”，这种生成的变化往往会给开发者尤其是初学者带来显著的理解上的困难。

虽然现有的一些相关的机器翻译测试技术[58]提出用蜕变测试对机器翻译的效果进行检测，然而，其并不是为程序生成所设计的。同时，即使检测出问题，该类机器翻译测试工作无法自动修复所检测出来的问题，进而难以达到自动提升程序生成生成效果的目的。因此，程序生成系统需要一个自动提升生成效果的技术以解决以上问题。

表 1.1　程序生成模型的错误示例

输入自然语言描述	**输出代码**

returns the first child element of a node which matches the given tag name.	T function (T arg0, String arg1) {
	for (T loc0: children()) {
	if (arg0.getTagName().equals(arg1)) {
	return loc0;
	}
	}
	return null;
	}

returns the first child element of a node that matches the given tag name.	T function (Document arg0, String arg1) {
	return (T) arg0.getFirstChild(arg1);
	}

1.3　本文的研究内容

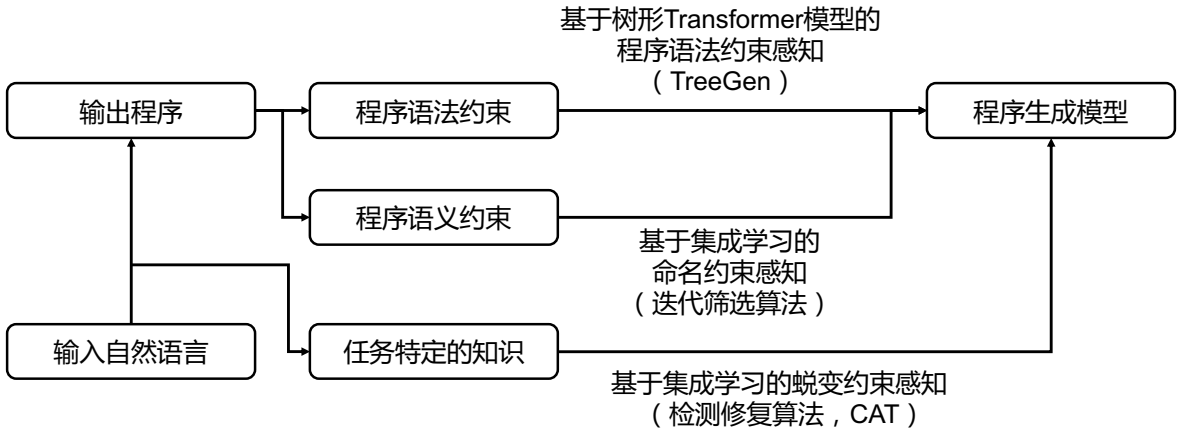

图 1.4　本文所提方法的整体概括

本论文工作针对现有工作的尚待解决的问题，围绕程序生成中的程序语法约束、语义约束、任务特定的约束三类约束的感知问题，分别提出其相应的方法（如图 1.4）。其主要目标是提升在程序生成任务下的神经网络模型感知约束的能力，从而进一步提升神经网络完成程序生成任务的能力，于实际开发过程中辅助开发者进行开发。

对于语法约束的感知上，本文设计了一种基于 Transformer 模型的程序语法约束感知方法（第1.3.1节）。

对于语义约束的感知上，本文针对变量、函数名等标识符命名约束，提出一种基于集成学习的命名约束感知方法。该方法以辅助神经网络训练的形式使得程序生成模型可以感知到命名约束（第1.3.2节）。

对于任务特定约束的感知上，本文将一种任务特定的约束表示为一种扰动约束。针

对该扰动约束，本文进一步提出了一种基于集成学习的扰动约束感知方法，以自动化检测和修改程序生成模型所生成的程序（第1.3.3节）。

1.3.1 基于 Transformer 模型的程序语法约束感知

程序生成系统基于输入自然语言描述的代码生成目标编程语言的代码。最现有的方法依赖于神经网络技术进行程序生成。然而，这些程序生成器在感知语法约束时存在上文所述尚待解决的问题。

本文提出了一种新的基于树的神经网络架构，TreeGen，用于程序生成。为了解决现有研究的问题，TreeGen 在玲珑框架[59]的基础上，使用 Transformers 的注意力机制来缓解长程依赖问题。该机制可以使得每个抽象语法树上的节点直接与其他节点进行计算，从而从根本上缓解了由神经网络循环计算所带来的长程依赖问题。同时，TreeGen 引入了一种新的具有树形结构的抽象语法树阅读器。该结构将语法规则信息和抽象语法树结构信息结合，使用图神经网络技术将抽象语法树表示为树结构，从而避免其被以“扁平”化的序列表示，实现了语法约束的结构编码。

本文在 Python 上的标准数据集 HearthStone 和两个语义解析标准数据集 ATIS 和 GEO 上评估了 TreeGen 模型的效果，TreeGen 在 HearthStone 上的表现比之前的最好的方法高出 9 个百分点，并且在 ATIS （89.1%）和 GEO （89.6%）数据集上取得了基于神经网络的方法的最佳生成准确率。

1.3.2 基于集成学习的命名约束感知

在语义约束感知方面，本文关注于变量、函数名命名关系问题。现有工作难以感知变量、函数名命名在程序语义上的性质，从而对程序生成的效果造成了影响。

为了感知变量、函数名的性质并使得神经网络可以既适应不同的变量、函数名命名风格又可以在命名质量较低的程序生成数据集上达到一定的训练效果，一种合理的想法是使用随机生成算法。该算法可以在不影响语义的情况下对变量、函数名进行随机交换，并使用交换后的数据进行训练。该方法由于随机命名的因素，可以让程序生成适应不同的变量、函数命名风格。

然而，如果直接通过随机交换的方式改变变量、函数命名会使得其生成的程序同样具有随机命名的风格。该命名风格难以被开发者所理解。同样的，该命名风格的程序在作为训练数据时往往不符合一般开发者的命名习惯，进而成为数据当中离群点，影响程序生成模型的训练。为了解决这个问题，本文进一步提出了一种迭代筛选算法，该技术在随机生成算法的基础上可以让神经网络自动找出变量、函数命名的最优组合。其通过一种无监督的方式让神经网络学会如何为变量、函数名分配合适的名称以完成生成程序的任务。

本文将迭代筛选算法与 TreeGen 模型相结合。在 Github 上所爬取的数据集上的实验结果表明，本文所提出的迭代筛选算法相比于不使用迭代筛选算法的技术生成程序的准确率提高了 8 个百分点。

1.3.3 基于集成学习的扰动约束感知

程序生成是一种从输入自然语言描述输出到目标程序语言代码的任务。现有的程序生成工作忽视了该任务特定的输入输出之间理应满足的扰动约束，从而对程序生成系统所生成程序的可接受性①造成了影响。

为了解决尚待解决的问题，本文需要设计一个能够感知扰动约束的方法，以达到提升程序生成系统所生成程序的可接受性的目的。本文认为程序生成系统应该满足以下扰动约束：输入的少量修改理应对输出的程序代码造成微小的影响（本文称之为一致性）。如果输入的少量修改对程序生成系统所生成程序造成了巨大的影响，本文则认为其是一个约束不一致性问题，而感知扰动约束的目标则是通过修复该不一致性问题以提升所生成程序的可接受性。

在这个关系的基础上，本文提出了一种感知上下文的检测和修复算法（Context-Aware Testing and repair for code generation，CAT），这是一种用于全自动检测和提升程序生成系统所生成程序的可接受性的方法。

CAT 将变异与蜕变测试相结合以实现自动检测程序生成系统中的不一致性问题的功能。然后，CAT 采用一种集成学习方法对检测出的问题进行自动修改。该技术生成大量具有相似输出的程序，利用输入输出一致性性质对原本的输入所对应生成的程序以集成学习的方式进行修改，从而实现所生成程序的可接受性的提升。

本文在常用的程序生成工具上对 CAT 进行验证。CAT 的自动检测方法发现：在验证模型上，大约有 39% 的输入存在约束不一致性问题。而 CAT 的黑盒修复平均为验证修复了 42% 的问题。灰盒修复平均为验证修复了 33% 的问题。该实验表明，CAT 使得程序生成方法感知到了所定义的扰动约束，进而达到提升程序生成方法所生成程序的可接受性的目的。

1.3.4 迭代筛选算法和检测和修复算法的拓展应用

本文所研究约束感知问题并不完全局限于程序生成。因而，本文所研究的约束感知方法也能用于其他存在同类约束的问题中。针对程序中的约束感知，本文对于命名约束感知提出了一种迭代筛选算法，对于扰动约束感知提出了一种检测和修复算法 CAT。

① 可接受性表示人工对程序生成模型效果的评价。可接受性越高，则程序生成系统所生成程序对人类帮助就越大。

这两种约束感知问题从本质上来说可以看作是两种通用问题，因为这两种约束往往也存在于众多相似的任务中。

命名约束感知问题本质上是对于一条数据中的名称等元素作替换不会影响最终结果的感知问题。举例来说，在图结构数据中当中常常存在一种无属性节点，该节点的名称不具有任何实际意义，而对其名称进行修改也不会影响其对应数据的含义。当图上所有的节点都是无属性节点时，该图则是一个无属性图。为了验证本文所提出的用于命名约束感知的迭代筛选算法的通用性，本文将迭代筛选算法推广到无属性图分类任务之上。该方法在无属性图分类中的布尔可满足性问题求解 (SAT) 及最大独立集求解任务上进行验证，其预测错误率相比于各自任务对应的现有最佳方法下降了 76%（布尔可满足性问题求解）和 39%（最大独立集求解）。

扰动约束感知问题本质上对于多条相近数据中应该具有某种关联关系的感知问题。该扰动约束往往也存在于各种生成任务当中。在机器翻译当中，对输入做某种相近变换，其对应的翻译也应需要做出相应的相近变换。为了验证本文所提出的用于扰动约束感知的检测和修复算法（CAT）的通用性，本文将 CAT 推广到机器翻译提升任务之上并在谷歌翻译及 Transformer 翻译器上进行验证。其检测技术发现在谷歌翻译和 Transformer 模型上大约有 40% 的约束不一致性问题。其修改技术平均为谷歌翻译和 Transformer 模型修复了 53% 和 52% 的问题。从提升翻译的角度，CAT 的修复在 16% 的情况下都具有更好的翻译可接受性（7.5% 更差）。

1.4 论文组织

本文组织分为六个部分，具体如下：

- **第一章 引言**。阐述本文工作的工作动机，总结本工作的研究背景和国内外相关的研究工作，明确需要研究的问题，在问题的基础上，进一步介绍本文的主要研究内容。
- **第二章 基于 Transformer 模型的程序语法约束感知**。本章针对于现有程序生成工作在语法约束感知方面的问题，借助于抽象语法树的树形语法结构信息及 Transformer 模型的能力，提出一种针对程序语法约束编码问题的树形 Transformer 模型。该模型可以编码树形的抽象语法树，并结合 Transformer 模型以解决编码中所面临的一系列问题，使得神经网络可以学习到程序中的语法约束，提升神经网络生成程序的能力。
- **第三章 基于集成学习的命名约束感知**。本章针对于现有程序生成工作在语义约束感知方面的命名约束感知问题，借助于集成学习的思想，本文提出了一种将集成学习和神经网络结合的方法，用以对命名约束进行编码，使得神经网络可

以学到程序中变量、函数名所对应的性质。

- **第四章 基于集成学习的扰动约束感知**。本章从扰动约束感知入手，提出了一种检测和修复算法，通过自动化的方式去检测模型对程序生成系统的扰动约束感知问题，同时，针对所检测出来的问题，提出了一种自动修复的方法解决该问题，进而提升程序生成系统所生成程序的可接受性。
- **第五章 迭代筛选算法和检测和修复算法的拓展应用**。本文所研究的约束感知问题也存在于无属性图分类领域及机器翻译领域。因而，本章将本文所提出的方法推广到这两个领域并设计实验进行验证本文所提出方法的有效性。
- **第六章 结论及展望**。对本文研究工作进行总结与展望。

第二章　基于 Transformer 模型的程序语法约束感知方法

本章提出一种结合语法约束的新型神经网络架构 TreeGen 用以完成程序生成任务。TreeGen 神经网络架构一共由三部分组成：(1) 自然语言阅读器（编码器），用以对输入的文本描述进行编码；(2) 抽象语法树阅读器（Transformer 解码器的前几层模块），在传统 Transformer 模型的基础上，在每个模块中额外使用了树卷积子层对已经生成的部分代码进行编码；(3) 一个解码器（其余的 Transformer 解码器模块），结合查询信息（抽象语法树中待扩展的节点）和前两个阅读器的输出结果来预测下一个语法规则。

为了解决现有工作面临的长程依赖问题和语法结构问题(详细细节请见第1.2.4.1节)，TreeGen 所使用的 Transformer 模型能够很好地捕捉长程依赖信息，从而在一定程度上缓解长程依赖的问题。不像现有工作利用具有较为严重的长程依赖问题的自回归网络去编码一段程序，Transformer 通过子注意力机制让程序中每个元素都可以直接与其他元素进行计算，进而缓解了该问题。同时，TreeGen 使用了一种特殊的抽象语法树阅读器，该阅读器将抽象语法树、图神经网络技术和 Transformer 模型三者结合，用以解决现有工作难以很好感知语法结构的问题。

2.1　程序生成方式

由于 TreeGen 模型依赖于程序生成的生成方式，因而本节介绍如何生成一段程序。根据现有工作的框架[59]，程序生成可建模为一系列语法规则的分类问题。

在程序语言中，一个程序可以被分解为一系列上下文无关的语法规则，并被解析器解析为一颗抽象语法树（Abstract Syntax Tree，AST）。举例来说，图 2.1 上显示了一段 Python 代码 “length = 10” 的 AST，其中虚线框表示的是抽象语法树的终结符（叶子节点），而实心框则是抽象语法树的非终结符（非叶子节点）。

基于 AST 的程序生成的基本过程是反复通过语法规则扩展已生成抽象语法树中的非终结符（如果有多个非终结符，TreeGen 会优先扩展最左下方的非终结符），直到所有的叶子节点都是非终结符为止。在图 2.1中，“*1: root -> Module*’’ 是一个语法规则的具体例子，前面的数字表示的是规则的编号。通过在树上的前序遍历，本章可以获得如该图右上角所示的一串语法规则序列。

从形式上来说，生成程序的概率可以分解为按照既定生成顺序使用语法规则的概率乘积（即最大似然估计值）。

$$p(\text{code}) = \prod_{i=1}^{P} p(r_i \mid \text{NL input}, r_1, \cdots, r_{i-1}) \tag{2.1}$$

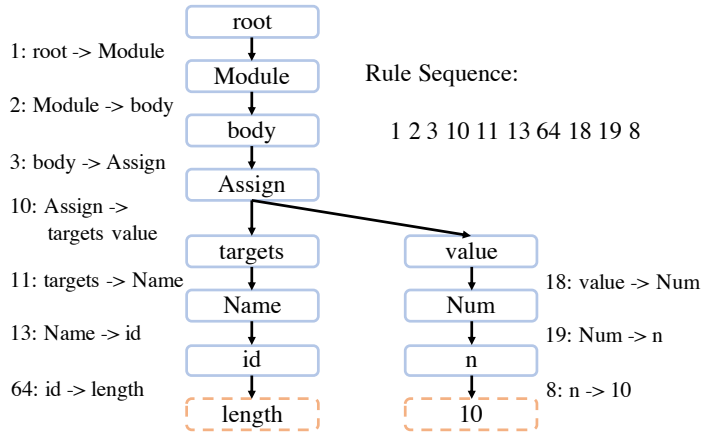

图 2.1　Python 代码 “length = 10’’ 的抽象语法树

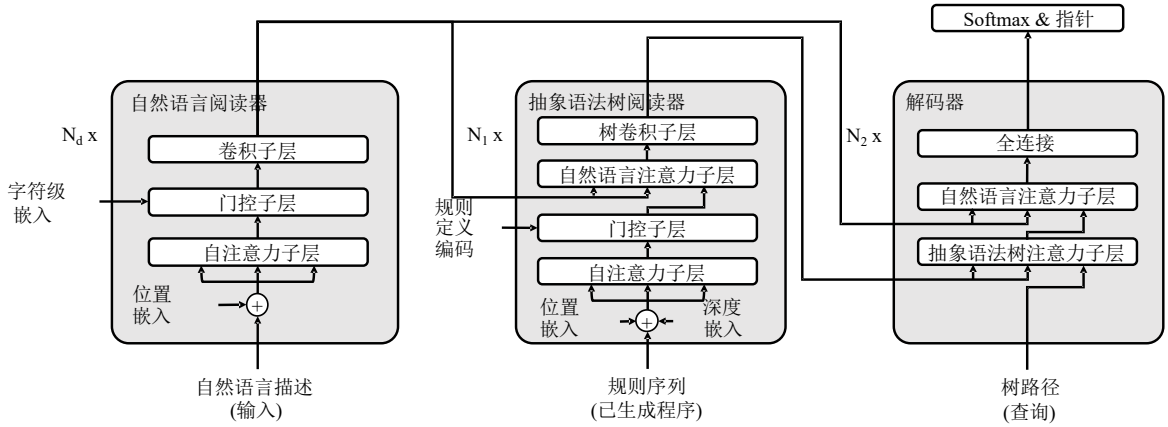

图 2.2　TreeGen 模型总览

其中 r_i 表示的该是规则序列中的第 i 个规则。从这个公式可以得出，本章需要完成的任务是训练一个神经网络模型来计算 $p(r_i \mid \text{NL input}, p_i)$。即，给定输入的自然语言描述和当前已生成的部分 AST，通过神经网络模型来计算下一步扩展指定节点所使用的规则。

2.2　程序语法约束感知方法

本章所提出的模型通过预测程序的语法规则来生成代码。图 2.2展示了 TreeGen 模型，它包括三个部分：自然语言阅读器、抽象语法树阅读器和解码器。本章将在以下小节中详细介绍它们的细节。

2.2.1　自然语言阅读器

输入的自然语言描述决定了程序所需要实现的具体功能。

对于一段输入描述，TreeGen 先通过分词的方式将其分为一串词序列 $n_1 n_2 \cdots n_L$，其中 L 表示输入的长度。然后将每个词 n_i 进一步拆分为其对应的字符序列 $c_1^{(n_i)}, c_2^{(n_i)}, \cdots, c_S^{(n_i)}$，其中 S 表示的是 n_i 中字符的长度。TreeGen 将所有的词和字符都通过嵌入的方式表示为实数向量 $\boldsymbol{n}_1 \boldsymbol{n}_2 \cdots \boldsymbol{n}_L$ 和 $\boldsymbol{c}_1^{(n_i)}, \boldsymbol{c}_2^{(n_i)}, \cdots, \boldsymbol{c}_S^{(n_i)}$。

2.2.1.1 输入文本表示

字符级嵌入　相似的词在很多情况下具有相似的字符（例如，“program” 和 “programs”）。这个属性在自然语言当中十分常见。为了利用这个属性，TreeGen 使用一个全连接层的字符级的嵌入来将其转化为字符级的向量表示

$$\boldsymbol{n}_i^{(\mathrm{c})} = W^{(\mathrm{c})}[\boldsymbol{c}_1^{(n_i)}; \cdots; \boldsymbol{c}_M^{(n_i)}] \tag{2.2}$$

其中 $W^{(\mathrm{c})}$ 是全连接层的权重。在本章中，TreeGen 将字符序列填充到一个预定义的最大长度 M。在该全连接层之后，TreeGen 还应用了层归一化（Layernorm）[60]以辅助训练。在得到字符级的向量表示之后，TreeGen 将这些向量输入到自然语言阅读器，并通过一个门控子层将其与词级别嵌入进行集成。

2.2.1.2 自然语言阅读器的神经网络结构

自然语言阅读器由一系列相同结构的模块组成（总共 N_d 个模块）。每个模块包含三个不同的神经网络子层（即自注意力子层、门控子层和词卷积子层）来提取输入描述的特征。其详细内容本章将在以下小节中介绍。特别的，和 Transformer 模型一样，TreeGen 在该模块不同的两个子层之间，使用了残差连接[61]和层归一化技术。

自注意力子层　自注意力子层是 Transformer 模型架构中的基础子层[62]，该子层通过使用多头注意力机制来捕获长程依赖信息。对于输入的词序列 $n_1, n_2, \cdots, n_L$，TreeGen 通过查找嵌入表格的方式将它们表示为嵌入向量 $\boldsymbol{n}_1, \boldsymbol{n}_2, \cdots, \boldsymbol{n}_L$。同时，TreeGen 使用了位置嵌入来编码词的位置信息。其位置嵌入采用了 Dehghani et al. (2018)[63]所提出的方法，并将第 b 个模块中第 i 个单词的位置嵌入计算为

$$p_{b,i}[2j] = \sin((i+b)/(10000^{2j/d})) \tag{2.3}$$

$$p_{b,i}[2j+1] = \cos((i+b)/(10000^{2j/d})) \tag{2.4}$$

其中 $p_{b,i}[\cdot]$ 表示向量 $\boldsymbol{p}_{b,i}$ 的第几维，d 则表示其维度大小（即嵌入向量的长度）。

TreeGen 通过多头注意力学习非线性特征，进而得到一个矩阵 $Y_b^{(\mathrm{self})} = [\boldsymbol{y}_{b,1}^{(\mathrm{self})}, \boldsymbol{y}_{b,2}^{(\mathrm{self})}, \cdots, \boldsymbol{y}_{b,L}^{(\mathrm{self})}]^\top$，其中 $Y_b^{(\mathrm{self})} \in \mathbb{R}^{L \times d}$。为了简化符号表示，本章省略了下标 b。其注意力中的

多头层可以由下式计算得到

$$Y^{(\text{self})} = \text{concat}(head_1, \cdots, head_H)W_h \tag{2.5}$$

其中 H 表示头的数量，W_h 是训练的权重。在每个头 $head_t$ 中，TreeGen 使用了一个由下式计算的自注意力机制

$$head_t = \text{softmax}(\frac{QK^\top}{\sqrt{d_k}})V \tag{2.6}$$

其中 $d_k = d/H$ 表示每个特征向量的长度。Q、K 和 V 由下式计算

$$[Q, K, V] = [\boldsymbol{x}_1, \cdots, \boldsymbol{x}_L]^\top [W_Q, W_K, W_V] \tag{2.7}$$

其中 $W_Q, W_K, W_V \in \mathbb{R}^{d \times d_k}$ 是模型参数。$\boldsymbol{x}_i$ 是该模块的输入。在第一个模块中，其输入是词级别嵌入和位置嵌入的向量和，即 $\boldsymbol{n}_i + \boldsymbol{p}_{1,i}$; 在后续模块中，它是前一个模块的输出和对应于该块的位置嵌入的向量和。

门控子层 在通过自注意力计算特征之后，本章进一步将其与字符级嵌入信息相结合。TreeGen 在这里使用了一种基于 softmax 的门控机制。对于第 i 个词，TreeGen 通过线性变换的方式从 $\boldsymbol{y}_i^{(\text{self})}$ 计算得到一个控制向量 $\boldsymbol{q}_i$。TreeGen 通过公式 2.2 中的 $\boldsymbol{n}_i^{(\text{c})}$ 线性变换得到字符级嵌入的向量 $\boldsymbol{k}_i^{(\text{c})}$ 。而，向量 $\boldsymbol{k}_i^{(\text{y})}$ 由 $\boldsymbol{y}_i^{(\text{self})}$ 通过另一个线性变换得出。其后续所使用的门控子层通过下式计算

$$[\alpha_{i,t}^{(\text{y})}, \alpha_{i,t}^{(\text{c})}] = \text{softmax}\{\boldsymbol{q}_i^\top \boldsymbol{k}_i^{(\text{y})}, \boldsymbol{q}_i^\top \boldsymbol{k}_i^{(\text{c})}\} \tag{2.8}$$

它们用于权衡 TreeGen 的特征向量 $\boldsymbol{v}_i^{(\text{y})}$ （由 $\boldsymbol{y}_i^{(\text{self})}$ 通过线性变换得到）和字符嵌入的特征向量 $\boldsymbol{v}_i^{(\text{c})}$（由 $\boldsymbol{n}_i^{(\text{c})}$ 通过线性变换得到）的权重。

$$\boldsymbol{h}_{i,t} = [\alpha_{i,t}^{(\text{y})} \boldsymbol{v}_i^{(\text{y})} + \alpha_{i,t}^{(\text{c})} \boldsymbol{v}_i^{(\text{c})}] \tag{2.9}$$

类似于公式 2.5，门控机制所得到的输出是 $Y^{(\text{gate})} = (\boldsymbol{h}_{i,t})_{i,t}$，其中 $(\cdot)_{i,t}$ 表示矩阵 $h_{i,t}$ 中的第 i 行 t 列的元素。

词卷积子层 最后，TreeGen 将两个卷积神经网络层应用于门控机制 $\boldsymbol{y}_1^{(\text{gate})}, \cdots, \boldsymbol{y}_L^{(\text{gate})}$ 的输出之上，用以提取每个词周围的局部特征。其输出为 $\boldsymbol{y}_1^{(\text{conv},l)}, \cdots, \boldsymbol{y}_L^{(\text{conv},l)}$，其中 l 表示卷积层的层数。$\boldsymbol{y}_i^{(\text{conv},l)}$ 由下式计算

$$\boldsymbol{y}_i^{(\text{conv},l)} = W^{(\text{conv},l)} [\boldsymbol{y}_{i-w}^{(\text{conv},l-1)}; \cdots; \boldsymbol{y}_{i+w}^{(\text{conv},l-1)}] \tag{2.10}$$

其中 $W^{(\text{conv},l)}$ 是卷积层中的可训练权重，$w=(k-1)/2$，k 表示窗口大小。特别的，$\boldsymbol{y}_i^{(\text{conv},0)}$ 表示门控子层的输出 $\boldsymbol{y}_i^{(\text{gate})}$。在这些卷积层中，TreeGen 使用了可分离卷积技术[64]。其原因是可分离卷积的参数较少，易于训练。对于第一个和最后一个词，TreeGen 使用了零填充技术。在这些层之间，TreeGen 使用了 GELU 激活函数[65]。

综上所述，自然语言阅读器通过多个模块的自注意力子层、门控子层和词卷积子层使得输入的自然语言描述被编码为特征向量 $\boldsymbol{y}_1^{(\text{NL})},\boldsymbol{y}_2^{(\text{NL})},\cdots,\boldsymbol{y}_L^{(\text{NL})}$。

2.2.2 抽象语法树阅读器

TreeGen 通过一个抽象语法树阅读器来对已经生成的部分 AST 的结构进行建模。虽然程序是通过预测语法规则序列来生成的，但这些规则本身缺乏程序的具体信息，难以用来预测下一条规则。因此，TreeGen 的抽象语法树阅读器更多地考虑了程序特定的语法约束，其中包括预测规则的规则信息和抽象语法树的树结构。

为了合并这些程序特定的信息，TreeGen 先将代码表示为一串语法规则序列，然后使用注意力机制对规则进行编码，最后使用树卷积层将每个节点的编码表示与其祖先的编码表示相结合。

2.2.2.1 抽象语法树表示

规则序列嵌入　为了对规则进行编码，TreeGen 使用规则的编号作为主要信息。假设本章有一串规则序列 $r_1,r_2,\cdots,r_P$ 用于在生成的步骤中生成部分 AST，其中 P 表示序列的长度。TreeGen 通过查找嵌入表格的方式将这些规则表示为实数向量序列 $\boldsymbol{r}_1,\boldsymbol{r}_2,\cdots,\boldsymbol{r}_P$。

规则定义编码　上述查表嵌入将语法规则视为原子标记（即只使用了规则编号信息），并丢失了规则的详细内容的信息。为了缓解这个问题，TreeGen 通过规则定义编码技术来增强规则信息的表示。

对于一个语法规则 $i:\alpha\rightarrow\beta_1\cdots\beta_K$，其中 α 表示的是父节点，$\beta_1\cdots\beta_K$ 表示的是其对应的子节点，子节点既可以是终结符也可以是非终结符，索引 i 表示是规则的编号。

类似于公式 2.2，TreeGen 通过一个全连接层将规则内容信息编码为向量 $\boldsymbol{r}^{(\text{c})}$，其输入是通过查表嵌入的方式得到父节点和子节点的嵌入向量 $\boldsymbol{\alpha},\boldsymbol{\beta}_1,\cdots,\boldsymbol{\beta}_K$。特别的，该序列也和之前的字符级嵌入计算一样，被填充到最大长度。

规则定义编码的特征向量 $\boldsymbol{y}_1^{(\text{rule})},\cdots,\boldsymbol{y}_P^{(\text{rule})}$ 由另一个全连接层计算得到

$$\boldsymbol{y}_i^{(\text{rule})}=W^{(\text{rule})}[\boldsymbol{r}_i;\boldsymbol{r}^{(\text{c})};\boldsymbol{\alpha}] \tag{2.11}$$

其中 $\boldsymbol{r}_i$ 是规则 r_i 的编号的查表嵌入，$\boldsymbol{r}_i^{(\text{c})}$ 是规则内容编码表示，同时，TreeGen 再次将

父节点的向量 α 输入到该层中。最后，如 Transformer 一样，TreeGen 使用了层归一化技术。

位置和深度嵌入 由于 TreeGen 的抽象语法树阅读器使用了自注意力机制，因而本章仍需要通过位置嵌入的方式来表示已使用的语法规则的具体位置。

TreeGen 采用公式 2.4 中的位置嵌入表示规则序列 $r_1, \cdots, r_P$ 中规则的使用顺序。其位置嵌入向量由 $\boldsymbol{p}_1^{(\mathrm{r})} \cdots, \boldsymbol{p}_P^{(\mathrm{r})}$ 表示。

然而，仅仅只依赖时间顺序上的位置嵌入信息，并不能让神经网络意识到其在抽象语法树中的位置信息。因而，为了编码这类位置信息，本章提出使用深度嵌入的技术。对于一个规则 $r : \alpha \rightarrow \beta_1 \cdots \beta_K$，TreeGen 用它的父节点在抽象语法树中的深度（即 α 的深度）表示其规则的深度。TreeGen 将其规则序列的深度通过查表嵌入的方式表示成一个深度嵌入序列 $\boldsymbol{d}_1, \cdots, \boldsymbol{d}_P$。

最后，TreeGen 将这两种不同的嵌入信息（位置和深度嵌入）与神经网络所得规则序列向量相加，从而使其信息获得整合。

2.2.2.2 抽象语法树阅读器的神经网络结构

抽象语法树阅读器和自然语言阅读器一样，由一系列结构相同的模块组成（总共 N_1 个模块）。每个模块可以进一步被分解为四个不同的子层（即自注意力子层、门控子层、自然语言注意力子层和树卷积子层）。如 Transformer 一样，除了树卷积子层外，TreeGen 在每个子层前后都采用了残差连接。同时，在每个子层之后，TreeGen 应用了层归一化。

自注意力子层 为了捕捉抽象语法树的整体信息，TreeGen 构建了一个类似与自然语言阅读器中的自注意力层，其中输入则是规则嵌入、位置嵌入和深度嵌入的总和，即 $\boldsymbol{r}_i + \boldsymbol{d}_i + \boldsymbol{p}_i^{(\mathrm{r})}$。自注意力子层使用了与公式 2.4, 2.5, 2.6 相同的机制提取特征 $\boldsymbol{y}_1^{(\text{ast-self})}, \boldsymbol{y}_2^{(\text{ast-self})}, \cdots, \boldsymbol{y}_P^{(\text{ast -self})}$。不同的是，这里的自注意力子层与其相比具有不同的权重，但添加了额外的深度嵌入到 $\boldsymbol{p}_i^{(\mathrm{r})}$ 之中。

门控子层 本章希望将内容编码规则表示 $\boldsymbol{y}_i^{(\text{rule})}$ 合并到 TreeGen 提取的特征中。TreeGen 采用与自然语言阅读器中相同的门控子层（公式 2.8, 2.9），其通过门控子层后的特征变为 $\boldsymbol{y}_1^{(\text{ast-g})}, \boldsymbol{y}_2^{(\text{ast-g})}, \cdots, \boldsymbol{y}_P^{(\text{ast-g})}$。

自然语言注意力子层 在生成程序的过程中，神经网络需要知道自然语言如何描述程序内容的信息。为了给出这种信息，TreeGen 将自然语言阅读器和抽象语法树阅读器通

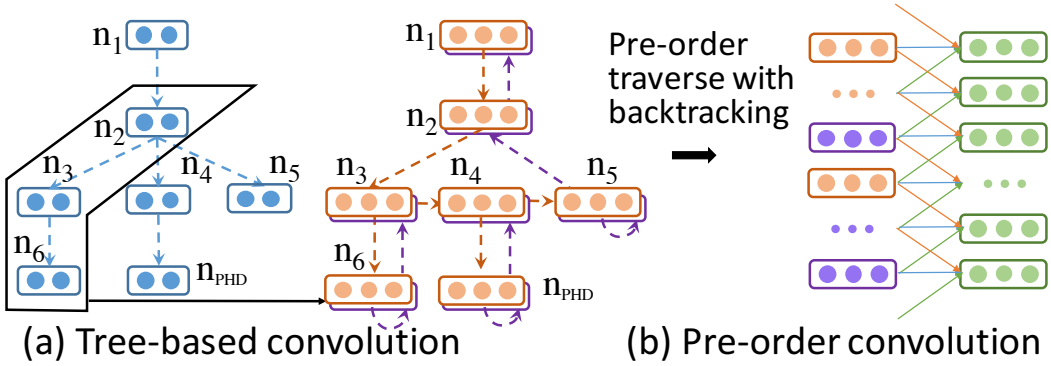

图 2.3　树卷积子层的详细计算过程

过多头注意力机制相连接。其计算方式类似于 Transformer 模型中解码器对其编码器的注意力机制[62]。TreeGen 将所提取到的特征记为 $\boldsymbol{y}_1^{(\text{ast-nl})}, \boldsymbol{y}_2^{(\text{ast-nl})}, \cdots, \boldsymbol{y}_P^{(\text{ast-nl})}$。

树卷积子层　如果只考虑上述子层，抽象语法树阅读器很难将语法节点的信息与其祖先结合起来（即很难考虑其语法结构上的信息）。一个语法节点可以在规则序列中离它所对应的父节点很远，同时，其在结构上又具有父子关系。对于这种问题，直接应用传统的 Transformer 模型很难提取出这样的结构化特征。

为了提取语法约束上的结构化的特征，TreeGen 将每个语法节点的特征与其父节点的特征相结合。为此，TreeGen 先将抽象语法树视为一个有向图，并进一步使用邻接矩阵的形式 M 来表示该有向图。其中，如果节点 α_i 是 α_j 的父节点，TreeGen 则设其对应的邻接矩阵中的值 $M_{ji} = 1$，否则，$M_{ji} = 0$。假设所有节点都由特征向量 $\boldsymbol{f}_1, \cdots, \boldsymbol{f}_n$ 表示，则它们的父节点对应的特征向量可以由其节点的特征向量序列与邻接矩阵相乘得到：

$$[\boldsymbol{f}_1^{(\text{par})}, \cdots, \boldsymbol{f}_n^{(\text{par})}] = [\boldsymbol{f}_1, \cdots, \boldsymbol{f}_n]M \tag{2.12}$$

其中 $\boldsymbol{f}_i^{(\text{par})}$ 表示第 i 个节点的父节点。其中，对于根节点的父节点，TreeGen 直接使用根节点本身的特征向量来对它进行填充。

最后，应用于当前已生成子树的树卷积子层的详细计算公式由下式给出（其计算过程如图 2.3）

$$\begin{aligned} Y^{(\text{tconv}, l)} = f(W^{(\text{tconv}, l)}[&Y^{(\text{tconv}, l-1)}; \\ &Y^{(\text{tconv}, l-1)}M; \cdots; Y^{(\text{tconv}, l-1)}M^{kt-1}]) \end{aligned} \tag{2.13}$$

其中 $W^{(\text{tconv}, l)}$ 是卷积层的可训练权重，kt 表示卷积的窗口大小（本章在实验中设置窗口大小为 3），l 是这些卷积层的层。特别的，$Y^{(\text{tconv}, 0)} = [\boldsymbol{y}_1^{(\text{att})}, \boldsymbol{y}_2^{(\text{att})}, \cdots, \boldsymbol{y}_P^{(\text{att})}]$，其中 $Y^{(\text{tconv}, 0)} \in \mathbb{R}^{d \times P}$。$f$ 表示 GELU 激活函数。

总的来说，抽象语法树阅读器具有 N_1 个相同的模块，每个模块都包含有上述四个

子层。最后一个模块输出的特征向量是 $\boldsymbol{y}_1^{(\text{ast})}, \boldsymbol{y}_2^{(\text{ast})}, \cdots, \boldsymbol{y}_P^{(\text{ast})}$。

2.2.3 解码器

TreeGen 的最后一个组件是一个解码器，它将已生成代码的信息与自然语言描述的信息描述相结合，并用以预测下一个所要使用的语法规则。与抽象语法树阅读器类似，在解码器中本章仍然使用一系列结构相同的模块（总共 N_2 个模块）。其中，每个模块中都包含有如下文所描述的多个结构相同的子层。在每个子层前后，TreeGen 也采用了残差连接和层归一化技术。

解码器将要扩展的非终结符的信息作为输入（本章称其为查询信息）。为了更好地便于神经网络理解输入的查询信息，TreeGen 使用树路径信息作为其具体输入。其树路径信息是一条由根节点到待扩展的节点的路径，该路径既包含了待扩展的节点信息也包含了该节点的结构信息。举例来说，如果要在图 2.1中扩展节点 “Assign”，那么其对应的树路径信息应该是 *root, Module, body, Assign*。TreeGen 将这个路径中的节点通过嵌入的方式表示为实数向量，然后对这些向量应用一个如公式 2.2中一样结构的全连接层，其对应树路径（查询节点）的输出是 $\boldsymbol{q}_i^{(\text{path})}$。

然后 TreeGen 应用两个注意力子层来整合抽象语法树阅读器和自然语言阅读器的输出向量。

TreeGen 先在抽象语法树阅读器的输出上应用抽象语法树注意力子层，并提取特征 $\boldsymbol{f}_1^{(\text{tree})}, \cdots, \boldsymbol{f}_P^{(\text{tree})}$。在这一层中，$Q$ 矩阵由查询 $\boldsymbol{q}_1^{(\text{path})}, \cdots, \boldsymbol{q}_P^{(\text{path})}$ 计算得出；K 和 V 矩阵则由代码特征 $\boldsymbol{y}_1^{(\text{ast})}, \cdots, \boldsymbol{y}_P^{(\text{ast})}$ 计算得出。更多的，本章进一步整合了输入自然语言描述中的特征。这种特征整合也是通过相似的自然语言注意力子层实现，其中 Q 矩阵由特征 $\boldsymbol{f}_1^{(\text{tree})}, \cdots, \boldsymbol{f}_P^{(\text{tree})}$ 计算得出；K 和 V 矩阵由输入描述 $\boldsymbol{y}_1^{(\text{NL})}, \cdots, \boldsymbol{y}_L^{(\text{NL})}$ 计算得出。

最后，TreeGen 使用了两个全连接层用于对所提取特征进行编码（在第一层中，TreeGen 使用了 $GELU$ 作为激活函数）。

2.2.4 训练和预测

TreeGen 对解码器的最后一层特征使用 softmax 激活函数以从所有可能的预定义语法规则候选中预测出下一个要使用的语法规则。

同时，TreeGen 还引入了指针网络机制[66]（其本质上是一个注意力机制）。该机制可以直接从输入的自然语言描述中复制一个指定位置的词 a。在这种情况下，其所使用的语法规则是 $\alpha \rightarrow a$，其中 α 是待扩展的非终结符，而 a 则是终结符。这种指针机制有助于生成用户自定义的标识符（例如，变量和函数名）。

预定义语法规则预测和指针网络规则语法之间的选择概率由另一个门控机制 p_g 计算得出。p_g 则是根据解码器的最后一层输出特征计算得到。总的来说，下一条要使

用的语法规则的预测概率为

$$p(r_i|\cdot) = \begin{cases} p_g\, p(r_i|\cdot) & \text{if } i \in \mathbf{D} \\ (1-p_g)\,\Pr\{\text{在第 } i \text{ 个预测步骤中拷贝单词 } t\,|\cdot\} & \text{if } i \in \mathbf{C} \end{cases} \tag{2.14}$$

其中 i 表示预定义规则的编号，$\mathbf{D}$ 是预定义规则的集合，$\mathbf{C}$ 表示指针网络所对应的拷贝规则集合（形式为 $\alpha \to a$，其中 a 是出现在自然语言描述中的终结符）。p_g（使用预定义规则类型的概率）和 $p(r_i|\cdot)$（每个预定义规则的概率）分别由不同的单层感知机计算得到，这两个单层感知机分别使用了 sigmoid 和 softmax 作为激活函数，其对应输入则是特征 $\boldsymbol{h}^{(\text{dec})}$。

其中，指针网络的具体运算如下式所示

$$\begin{aligned} \xi_t &= \boldsymbol{v}^T \tanh(W_1 \boldsymbol{h}^{(\text{dec})} + W_2 \boldsymbol{y}_t^{(\text{NL})}) \\ \Pr\{\text{在第 } i \text{ 个预测步骤中拷贝单词 } t\,|\cdot\} &= \frac{\exp\{\xi_t\}}{\sum_{j=1}^{L} \exp\{\xi_j\}} \end{aligned} \tag{2.15}$$

其中 $\boldsymbol{h}^{(\text{dec})}$ 表示解码器的最后一个特征。

TreeGen 模型的训练通过优化最大似然估计的损失函数来进行。

预测的过程由 *start* 规则开始（$start : snode \longrightarrow root$）。该规则表示将特殊非终结符 *snode* 扩展生成根节点 *root* 。如果预测的抽象语法树中的所有的叶子节点都是终结符，那么 TreeGen 则认为程序生成完毕。否则，本章会反复调用 TreeGen 模型对已生成的抽象语法树进行扩展。在预测过程中，本章使用大小为 5 的波束搜索算法，同时，在波束搜索时，TreeGen 通过过滤的方式排除了不合法的语法规则。

2.3　实验验证 1: Python 生成任务

本章先在 Python 标准测试数据集上中验证了本章所提出的 TreeGen 方法。

2.3.1　实验设置

模型设置　对于 TreeGen 模型，本章将自然语言阅读器的模块数设置为 6（即 $N_d = 6$）。同时，本章为抽象语法树阅读器和解码器设置了相同的模块数（即 $N_1 = N_2 = 5$）。在模型中，所有嵌入的维度大小为 256，所有隐含层大小也都设置为 256（特别的，对于每个全连接层，本章将其第一层的隐含层大小设置为 1024 维）。在每一层结束之后，本章还使用了 dropout 机制（其中 dropout 所对应的正则化概率为 0.15）。在训练的过程中，该模型由具有默认参数的 Adafactor[67] 优化器进行优化。

Darkscale Healer NAME_END
4 ATK_END
5 DEF_END
5 COST_END
-1 DUR_END
Minion TYPE_END
Neutral PLAYER_CLS_END
NIL RACE_END
COMMON RARITY_END
Battlecry: Restore 2 Health to all friendly characters.

```
class DarkscaleHealer(MinionCard ) :
    def __init__ (self) :
        super().__init__("Darkscale Healer", 5, CHARACTER_CLASS.ALL,
            CARD_RARITY.COMMON, battlecry =
            Battlecry(Heal(2), CharacterSelector(
                players = BothPlayer())))

    def create_minion (self, player) :
        return Minion(4, 5)
```

图 2.4　HearthStone 数据集的实现示例

数据集　本章在 HearthStone 数据集上中评估了 TreeGen 方法[6]。该数据集包含 665 种不同的炉石传说游戏卡牌的 Python 实现代码。其中，每张游戏卡牌则由一个半结构化输入自然语言描述和一个正确的 Python 程序组成。输入自然语言描述由多个不同的属性组成（其中包括卡牌名称、卡牌类型以及卡牌功能的自然语言描述等）。Python 程序的平均长度为 84 个元素。Python 程序主要实现由自然语言描述中的卡牌功能描述所决定，而其他属性则决定代码中的部分常量或者标识符的值。

图 2.4展示了一条该数据的示例（包括半结构化自然语言描述及其对应的 Python 程序）。由于输入是半结构化的，因而现有工作在将输入描述预处理为一系列词序列时考虑了两种方法。第一种方法[12,56]（称为纯文本预处理）将整个半结构化自然语言描述视为纯文本信息，并通过空格或标点等标准分隔符对文本进行分隔。第二种方法[55]（称为结构化预处理）将输入描述视为半结构化的语言，并将每一个属性独立视为一个单独的词语。

本实验同时考虑了这两种不同的处理方法，并将对应于纯文本预处理的结果表示为 TreeGen-A，将对应于结构化预处理的结果表示为 TreeGen-B。

在实验的过程中，按照 Ling et al. (2016) 中的训练集-验证集-测试集的分割方法对数据集进行划分，其对应的统计数据在表 2.1 中展示。

验证指标　本章通过程序生成准确率和 BLEU 分数这两种不同的验证指标来验证本章的 TreeGen 方法。在理想情况下，程序生成的准确率应该为所生成的功能正确的程序所占比例，不幸的是，功能正确的程序并不是图灵可计算的。因而，本章遵循了之前的

表 2.1　本章所使用的数据集所对应统计信息

	实验 2		
数据	**HS**	**ATIS**	**GEO**
# 训练集	533	4,434	600
# 验证集	66	491	-
# 测试集	66	448	280
自然语言描述中词语数量平均值	35.0	10.6	7.4
自然语言描述中词语数量最大值	76.0	48	23
程序中词数量平均值	83.2	33.9	28.3
程序中词数量最大值	403	113	144

大部分研究的方法[6,12]，并直接根据字符串匹配的方法来计算准确率（记为 **StrAcc**，如果生成的程序和标准的正确程序具有完全相同的字符序列，本章则认为其是正确生成的程序）。① 本章在验证过程中发现，几个所生成的程序虽然使用了不同的变量名称但却实现了正确的功能。虽然其与标准的正确程序不同，但经过人工检查是显然正确的程序。因而，本章用 **Acc+** 表示人工检查调整后的准确率。在这里，本章没有人工检查程序算法等的非显而易见的等价程序，因此 Acc+ 仍然是程序生成任务功能准确率的下界。

2.3.2　实验结果

实验的结果展示在表 2.2 中。在该表中，本章发现对输入半结构化自然语言描述使用结构化预处理相比于纯文本处理整体来说具有更好的效果。

如表中数据所示，TreeGen 在对输入描述进行纯文本处理的条件下，相比于现有的最佳方法实现了 6 个百分点的准确率提升，而在对输入描述进行结构化处理的条件下实现了 9 个百分点的准确率提升（CodeGPT 则是在 TreeGen 模型提出之后的方法）。对于 BLEU 分数来说，TreeGen 也取得了现有方法中最佳的效果。该实验表明 TreeGen 可以有效地感知语法约束从而提升了模型效果。

时间效率验证　本章进一步评估了 TreeGen 模型在 HearthStone 数据集上的时间效率。其对应的实验结果表明，TreeGen 模型比之前的模型训练更快。其在单个 Nvidia Titan XP 上使用完整的训练集训练一代需要 18 秒，RNN[12] 模型需要 49 秒。

① 由于抽象语法树中不表示空格和空行，因此本章这里的字符串匹配为基于程序“标准化”的格式计算。特别的是，抽象语法树可以隐式捕捉对于 Python 程序来说至关重要的缩进信息。

表 2.2　HearthStone 数据集的实验效果

	模型	StrAcc	Acc+	BLEU
纯文本	LPN[6]	6.1	–	67.1
	SEQ2TREE[54]	1.5	–	53.4
	YN17[12]	16.2	~18.2	75.8
	ASN[55]	18.2	–	77.6
	ReCode[56]	19.6	–	78.4
	TreeGen-A	**25.8**	**25.8**	**79.3**
结构化	ASN+SUPATT[55]	22.7	–	79.2
	CodeGPT[68]	27.3	30.3	75.4
	TreeGen-B	**31.8**	**33.3**	80.8
	树卷积子层的位置			
	$N_1 = 10, N_2 = 0$	25.8	27.3	80.4
	$N_1 = 10(7), N_2 = 0$	27.3	28.8	78.5
	$N_1 = 10(8), N_2 = 0$	25.8	28.8	78.5
	$N_1 = 0, N_2 = 10$	21.2	22.7	79.6
	消融实验			
	基准模型: Transformer	10.6 ($p = 0.015$)	12.1	68.0
	- 树卷积	27.3 ($p = 0.015$)	27.3	80.9
	- 规则定义编码	27.3 ($p < 0.001$)	28.8	**81.8**
	- 字符级嵌入	15.2 ($p < 0.001$)	18.2	72.9
	- 自注意力	28.8 ($p < 0.001$)	28.8	81.0

树卷积子层的位置验证　TreeGen 的关键之一是将树卷积子层仅添加到解码器中的部分 Transformer 模块中。为了评估这个设计是否有效，本章验证了四个不同的设置：1）将树卷积子层添加到所有 AST 阅读器模块中（即 $N_1 = 10$）；2) 将树卷积子层添加到 AST 阅读器的前 7 个模块中（即 $N_1 = 10(7)$）；3) 将树卷积子层添加到 AST 阅读器的前 8 个模块中（即 $N_1 = 10(8)$）；4) 不添加树卷积子层（即 $N_1 = 0$）。正如表 2.2中所示，TreeGen 将子层添加到所有模块中 ($N_1 = 10$) 优于最后一个设置 ($N_1 = 0$)，但略差于其他两个设置。

消融实验　本章设置了消融实验来进一步分析 TreeGen 模型（使用了 TreeGen-B）中每个组件对于模型效果的贡献程度。其实验结果展示在表 2.2 中。首先，本章将 TreeGen 模型与传统的 Transformer 进行了比较，传统的 Transformer 是一个没有有效语法约束感知机制的模型。实验结果表明，TreeGen 的准确率提高了 21 个百分点（p 值小于 0.001），BLEU 值提高了 12 个百分点。这一结果有力地证明了抽象语法树阅读器在本章的模型中的有效性以及语法约束感知的重要性。接下来，本章将抽象语法树阅读器中的树卷

积层替换为两层简单的全连接层，并依次移除了字符级别嵌入、规则定义编码、自注意力层。实验结果表明，这些层所对应的标识符编码、长程依赖问题和结构信息显著影响着模型生成程序的准确率。特别的，在某些情况下，在 BLEU 增加的同时，StrAcc 和 Acc+ 会减少。然而，StrAcc 和 Acc+ 这两个指标更为重要，因为它们保证了生成程序的正确性，一些看似正确却不正确的程序有时候会为开发者的调试过程带来麻烦。

生成示例　图2.5展示了两个来自 HearthStone 测试集的生成示例。

图中（a）表示 HearthStone 游戏卡牌中的一张怪兽卡，其右上角为输入半结构化的自然语言描述（分别是卡牌名，攻击力，防御力，花费，耐久，种类，玩家职业，种族，稀有度和效果），而图的下面则是由我们 TreeGen 模型所正确生成的 Python 代码。该卡牌的代码分为两个函数，第一个函数包括卡牌的基本属性和卡牌效果，其中卡牌效果（从 battlecry 开始的部分代码）所占代码比例较大且较其他属性来说更为复杂，第二个函数则是卡牌的另一部分基本属性。

图中（b）表示 HearthStone 游戏卡牌中的一张武器卡，其右上角为输入半结构化的自然语言描述，图的下方为 TreeGen 模型生成的错误代码。该卡牌代码的第二个函数较为复杂，其主要由输入中的卡牌描述生成。由于该类描述的训练数据较少，因而 TreeGen 没有正确生成其代码，然而，TreeGen 所生成的代码仅仅只比正确代码多了一个参数（condition）赋值。

2.4　实验验证 2: Lambda 演算生成任务

本章进一步在两种 Lambda 演算生成的标准数据集（ATIS 数据集和 GEO 数据集）上中验证本章所提出的 TreeGen 方法。

2.4.1　实验设置

模型设置　与 HearthStone 实验所使用的模型设置相比，本章将嵌入大小和隐含层大小更改为 128 维。除此之外，本章遵循了在 HearthStone 实验中的所有相同的模型设置。

数据集　本章在 Dong 和 Lapata（2016）[54] 中使用的两个 Lambda 演算数据集（ATIS 和 GEO）上验证了 TreeGen 模型效果。在这两个数据集中其每条数据的输入是自然语言描述，输出是一段 λ-演算的程序代码（如图 2.6 所示）。本章对数据集使用了标准的训练集-验证集-测试集的划分方法[69]。其统计数据列在表 2.1 中。相比于 HearthStone 数据集，ATIS 和 GEO 这两个数据集代码较短，难度较之有些许下降。

Hungry Dragon NAME_END
5 ATK_END
6 DEF_END
4 COST_END
-1 DUR_END
Minion TYPE_END
NEUTRAL PLAYER_CLS_END
DRAGON RACE_END
COMMON RARITY_END
< b > battlecry : < /b > summon a random 1 - cost minion for your opponent .

```
class HungryDragon(MinionCard ) :
    def __init__ (self) :
        super().__init__("Hungry Dragon", 4, CHARACTER_CLASS.ALL,
            CARD_RARITY.COMMON, minion_type = MINION_TYPE.DRAGON,
            battlecry = Battlecry(Summon(CardQuery(
            conditions = [ManaCost(1), IsMinion()])),
            PlayerSelector(EnemyPlayer())))

    def create_minion (self, player) :
        return Minion(5, 6)
```

(a) TreeGen 模型生成正确的数据

Death 's Bite NAME_END
4 ATK_END
-1 DEF_END
4 COST_END
2 DUR_END
Weapon TYPE_END
WARRIOR PLAYER_CLS_END
NIL RACE_END
COMMON RARITY_END
< b > deathrattle : < /b > deal 1 damage to all minions .

```
class DeathsBite(WeaponCard ) :
    def __init__ (self) :
        super().__init__("Death's Bite", 4, CHARACTER_CLASS.WARRIOR,
            CARD_RARITY.COMMON)

    def create_weapon (self, player) :
        return Weapon(4, 2, deathrattle = Deathrattle(Damage(1),
            MinionSelector( condition = IsType(MINION_TYPE.minions)
                , players = BothPlayer())))
```

(b) TreeGen 模型生成错误的数据

图 2.5　TreeGen 在 HearthStone 数据集上的生成示例

输入描述： list airport in ci0
输出 λ-演算程序：

```
lambda $0 e ( and ( airport $0 )
              ( loc:t $0 ci0 ) )
```

图 2.6　ATIS 数据集的实例

表 2.3　在 Lambda 演算数据集上的准确率

	方法	ATIS	GEO
传统技术	ZC07[70]	84.6	86.1
	FUBL[71]	82.8	88.6
	KCAZ13[72]	-	89.0
	WKZ14[73]	**91.3**	**90.4**
神经网络技术	SEQ2SEQ[54]	84.2	84.6
	SEQ2TREE[54]	84.6	87.1
	ASN[55]	85.3	85.7
	ASN+SUPATT[55]	85.9	87.1
	COARSE2FINE[11]	87.7	88.2
	TRANX[74]	86.2	88.2
	Seq2Act[75]	85.5	88.9
	Graph2Seq[76]	85.9	88.1
	SZM19[77]	85.0	-
	CodeGPT[68]	87.5	-
	TreeGen	**89.1**	**89.6**

验证指标　在这个任务中，本章遵循之前的方法[54]，并使用树准确率作为验证指标。如果一个生成的语法树与参考语法树相同（忽略一些由可置换的节点所造成的语法树差异），本章则认为其生成是正确的。最后，本章统计正确的程序占总程序的比例作为树准确率。

2.4.2　实验结果

表 2.3 显示了 TreeGen 和各对比模型的效果。可以看出，TreeGen 方法的准确性低于基于 CCG 解析器并使用大量人工定义模板的传统方法 WKZ14[73]。然而，这种传统方法很难泛化新数据集，相比之下本章的 TreeGen 模型和 HearthStone 实验中的模型具有相同的结构。

与其他使用神经网络模型的技术相比，TreeGen 在 ATIS 和 GEO 数据集上分别取得了 89.1% 和 89.6% 的树准确率。这表明，TreeGen 在所有神经网络模型中具有最好的效果[11,54,55,75-77]。该实验显示了 TreeGen 的有效性和通用性。

语法约束学习 为了进一步分析 TreeGen 是否感知到了语法约束，本节进一步在 ATIS 数据集上进行验证，如果预测的规则违反了 ATIS 标准数据集中规定的语法约束，则认为其预测错误，否则则认为其预测正确。实验结果表明，TreeGen 可以有 99% 的预测概率成功预测出合理的语法规则，这表明 TreeGen 成功学到了有效的语法约束。

2.5 小结

对于程序语法约束的感知，本章提出了 TreeGen 模型用于将程序语法约束和程序生成相结合。TreeGen 使用 Transformer 模型的注意力机制来缓解长程依赖问题，同时，TreeGen 引入了抽象语法树阅读器，将语法规则和抽象语法树结构结合起来。

本章的实验验证在 Python 数据集 HearthStone 和两个 Lambda 演算数据集 ATIS 和 GEO 上。实验结果表明，TreeGen 模型明显优于现有方法。同时，本章还进行了深入的消融实验，这表明 TreeGen 模型中所使用的每个组件都起着重要作用。

第三章　基于集成学习的命名约束感知

为了感知标识符命名约束，使得神经网络既可以适应不同的变量、函数名命名风格又可以在命名质量较低的程序生成数据集上达到一定的训练效果（第1.2.4.2节）。现有的工作并没有关注到该命名约束。一种简单的想法是使用一种随机生成算法。该算法通过随机的方式在不影响程序语义的情况下为程序中变量、函数名节点交换顺序从而获得不同命名风格的训练数据样本，来消除这些变量、函数名命名风格差异及质量高低所造成的影响。对于训练，神经网络每次处理数据样本时，该算法都会重新在不影响程序语义的情况下随机交换多个不同的变量、函数名节点。通过这种方式，该算法可以在不同的交换下使得变量、函数名之间的关联关系保持不变而具体的命名名称发生改变。进而强迫神经网络学会变量、函数名之间的关联关系。同时，变量、函数名命名风格及质量所造成的影响会被多次交换节点命名并训练所平滑。

这个方法虽然解决了现有技术的问题，但是引入了一个新的问题。其具体是：在生成的过程中，神经网络会倾向于像训练时一样随机选出一个变量、函数名当作最终的命名。然而，这样命名风格的程序往往也提升了开发者甚至神经网络自身理解程序语义的难度。同时，在训练过程中，这样随机生成的命名也会导致所得训练数据不符合一般开发者的命名习惯，进而成为数据当中离群点，影响程序生成模型的训练。举例来说，在图 3.1的程序中，开发者往往会将输入参数命名为 array（或者为与 array 相关命名，如拼音：shuzu），将 for 循环的参数命名为 i 和 j。而随机生成则会将改命名进行乱序，使得输入参数可能被命名为 i，而 for 循环的参数则变为 array。这样的数据在训练集中出现频次较低，因而可能会导致程序生成模型难以学习该数据。

为此，本章提出了一个更适用于感知变量、函数名命名约束的迭代筛选算法。该技术在训练期间仍旧为神经网络的每条训练数据进行多次变量、函数名节点交换，但是只使用具有最佳命名的数据对神经网络进行参数优化。通过这种方式，神经网络可以无监督地选择在训练中出现频次最高的命名顺序进行训练，从而避免了由随机生成命名所造成的训练影响。

```
def sort(array):
    for i in range(len(array)-1):
        for j in range(0, len(array)-i-1):
            if array[j] > array[j+1]:
              array[j], array[j+1] =
             array[j+1], array[j]
```

图 3.1　程序代码示例

3.1 基于迭代筛选的命名约束感知方法

本章先解决现有程序生成技术感知变量、函数名命名约束的问题。通过本章的进一步分析，本章提出了最终的迭代筛选算法。

Algorithm 1: 随机生成的训练过程的伪代码

```
Data: D: 训练集; E: 最大的训练代数; K: 随机采样次数; TREEGEN: 程序生成模型
epoch = 0
while epoch < E do
    epoch += 1
    for 每一个训练块 B ∈ D do
        dataList = []
        for B 中每一个自然语言描述 G 及其对应的参考程序 Y do
            lowestLoss = inf
            candidateG = None
            i = 0
            while i < K do
                i += 1
                // 对参考程序 Y 在不影响程序语义的情况下进行变量、函数名的随机交换
                Y = randomPermute(Y)
                // 将数据加入训练集
                data = (G, Y)
                dataList.append(data)
            end
        end
        // 使用随机生成后的数据来优化程序生成模型
        TREEGEN.optimize(dataList)
    end
end
```

3.1.1 随机生成

为了解决节点现有工作受制于训练数据的变量、函数名命名质量的问题，本章先提出了一个有一定合理性的随机生成算法。该算法通过随机的方法在不影响程序语义的情况下对程序中的变量、函数名进行交换以优化训练数据的效果[78,79]，如算法 1 和图 3.2所示。

对于训练来说，其程序中各变量、函数名节点的具体名称是随机分配得到的。这可以作为一种数据增强的方式，即通过集成学习的方式对数据做出平滑，在每个训练代中，随机生成算法只训练一种随机表示。这种平滑的方式会使得使用了随机生成算

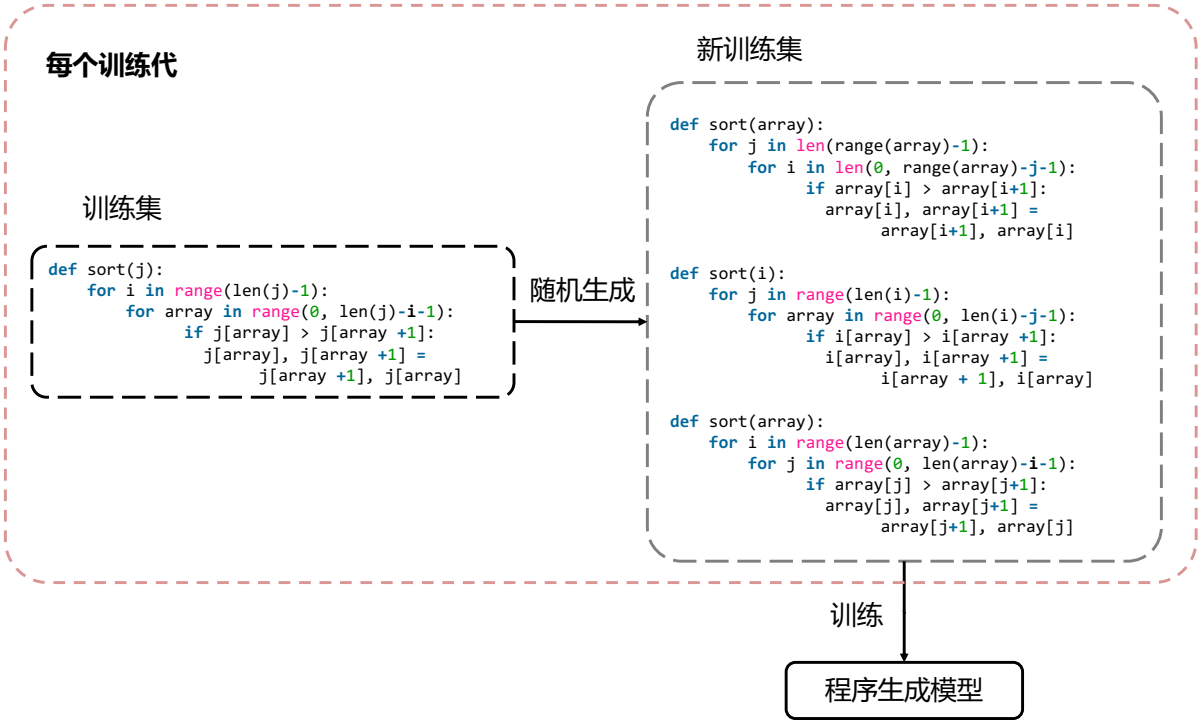

图 3.2　随机生成

法的模型所使用的训练数据中的命名质量高低也会获得平滑，从而减轻了命名质量问题对于程序生成技术的影响。

上述随机生成算法的伪代码如算法 1 所示。在该算法中，输入为一个训练集 D，最大的训练代数 E，随机采样次数 K 和程序生成模型 $TREEGEN$（本章拿 TreeGen 模型作为示例）。该算法先将一个用于记录训练代数的变量 epoch 设置为 0（第 1 行），然后对模型进行不断训练，直到训练代数 epoch 大于预定义的最大训练代数 E（第 2 行）。对于每一次训练来说，先将训练代数 epoch 增加 1（第 3 行）。对于训练集 D 中每一个训练块数据 B（第 4 行），随机生成算法先设定一个用于训练的最终数据集合 dataList 并将其置为空（第 5 行）。对于训练块 B 中的每一个待训练数据（第 6 行），随机生成算法对其进行 K 次修改（第 10 行和第 13 行）并将每次修改后的数据加入到新的训练集当中（第 15 行和第 16 行）。对每一条训练块中每一条数据都进行处理之后，随机生成算法使用所得的最终训练数据对模型进行优化（第 20 行）。

但是，如果直接将随机生成算法应用到程序生成任务中是不合适的。其训练所使用的交叉熵训练本质上是

$$\underset{\omega}{\text{minimize}} \sum_{(X,Y)\in\mathcal{D}} \sum_{\pi\in S_n} \sum_{i=1}^{n} D_{\text{KL}}\left((\pi(Y))_i \parallel f_i(\pi(X);\omega)\right),$$

其中 ω 表示 GNN 中的可训练参数，$\mathcal{D}$ 是训练集，D_{KL} 是预测和参考值之间的 Kullback-

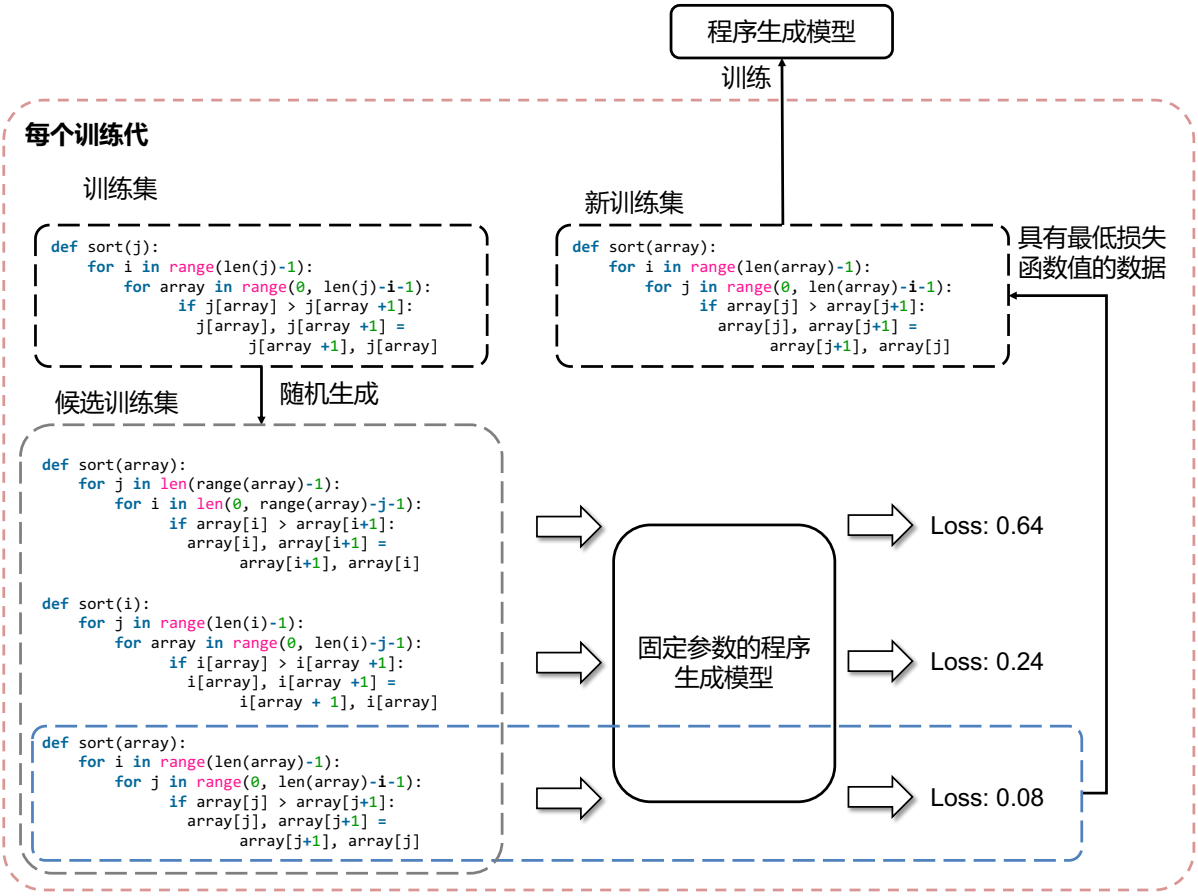

图 3.3　本章所提出的迭代筛选算法

Leibler 散度（KL 散度）。$(\pi(Y))_i$ 和 $f_i(\pi(X))$ 是矩阵中的第 i 行，表示随机生成的预测参考分布和实际预测分布。该训练目标会强制程序生成技术学习训练集中随机的变量命名风格。该随机的命名风格往往提升了开发者在使用程序生成技术时的理解难度。同时，也如在本章前文中所分析的那样，由于随机命名风格很少出现于由开源代码网站上所爬取的训练数据当中，换句话说，这种随机命名风格的程序是训练数据中所存在的离群点，而这样的离群点会使得程序生成技术在训练的过程中难以对其进行学习并生成。

为了解决该问题，本章进一步提出了一种迭代筛选算法。

3.1.2　迭代筛选算法

迭代筛选算法的概述如图 3.3所示。

对于训练来说，迭代筛选算法仍然为变量、函数名在不影响程序语义的情况下进行随机交换以生成新数据。为了使得所生成的程序摆脱随机的命名风格。迭代筛选算法会尽可能使得变量、函数命名与训练集中大部分程序的命名风格接近，迭代筛选算法在每个训练代中为输入的程序动态采样多个随机交换，但程序生成技术只训练具有

Algorithm 2: 迭代筛选算法训练过程的伪代码

```
Data: D: 训练集; E: 最大的训练代数; K: 随机采样次数; TREEGEN: 程序生成模型
epoch = 0
while epoch < E do
    epoch += 1
    for 每一个训练块 B ∈ D do
        dataList = []
        for B 中每一个自然语言描述 G 及其对应的参考程序 Y do
            lowestLoss = inf
            candidateY = None
            i = 0
            while i < K do
                i += 1
                // 对参考程序 Y 在不影响程序语义的情况下进行变量、函数名的随机交换
                Y = randomPermute(Y)
                // 将交换后的程序输入到程序生成模型并计算其损失函数
                y = TREEGEN(G)
                loss = computeLoss(y, Y)
                if loss < lowestLoss then
                    lowestLoss = loss
                    candidateY = Y
                end
            end
            // 选出具有最低损失函数值的数据
            data = (G, candidateY)
            dataList.append(data)
        end
        // 使用具有最低损失函数值的数据来优化程序生成模型
        TREEGEN.optimize(dataList)
    end
end
```

最佳命名的交换（即计算所得损失最低的）的数据。在下一个训练代中处理程序时，迭代筛选算法会重新在不影响程序语义的情况下随机交换变量、函数名并进一步重复此训练过程。

从形式上来说，迭代筛选算法在程序生成模型中将一个程序变量名、函数名根据其名称分别表示为两串不同的嵌入向量 $\boldsymbol{e}_1 \cdots \boldsymbol{e}_N$ 和 $\boldsymbol{f}_1 \cdots \boldsymbol{f}_Q$，其中 N/Q 是程序中所含有变量名/函数名的数量。在每个训练代中，本章随机采样两个交换 π 和 π_f，用于对数据中的变量、函数名进行处理（$\pi(\boldsymbol{e}_1 \cdots \boldsymbol{e}_N)$ 和 $\pi_f(\boldsymbol{f}_1 \cdots \boldsymbol{f}_N)$）。

本章重复这个采样过程 K 次，并计算这些交换在输入到程序生成模型中的损失函数值。最后，本章选择损失函数值最小的交换作为用于训练的最终程序命名。在不同的训练代中，即使对于相同的数据样本，这些交换也会被重新采样。

以上的训练过程可以描述为

$$\underset{\omega}{\text{minimize}} \sum_{(X,Y)\in\mathcal{D}} \min_{\pi\in S_n} \sum_{i=1}^{n} D_{\text{KL}}\left((\pi(Y))_i \,||\, f_i(\pi(X);\omega)\right) \tag{3.1}$$

上述迭代筛选算法的伪代码如算法 2 所示。在该算法中，输入为一个训练集 D，最大的训练代数 E，随机采样次数 K 和程序生成模型 $TREEGEN$（本章拿 TreeGen 模型作为示例）。该算法先将一个用于记录训练代数的变量 epoch 设置为 0（第 1 行），然后对模型进行不断训练，直到训练代数 epoch 大于预定义的最大训练代数 E（第 2 行）。对于每一次训练来说，先将训练代数 epoch 增加 1（第 3 行）。对于训练集 D 中每一个训练块数据 B（第 4 行），迭代筛选算法先设定一个用于训练的最终数据集合 dataList 并将其置为空（第 5 行）。对于训练块 B 中的每一个待训练数据（第 6 行），迭代筛选算法对其进行 K 次修改（第 10 行和第 13 行）。如果修改后的数据在程序生成模型当中具有更低的损失函数值，该伪代码会将其记录（第 15-19 行）。在 K 次修改之后，迭代筛选算法挑选出具有最低损失函数值的数据作为该训练代中的最终训练数据之一（第 23 和 24 行）。对每一条训练块中每一条数据都进行处理之后，迭代筛选算法使用所得的最终训练数据对模型进行优化（第 27 行）。

迭代筛选算法不受随机生成算法对于变量名随机命名的影响。因为迭代筛选算法在每次训练的过程中会无监督地使得变量、函数命名与训练集中大部分程序的命名风格接近。

3.2 实验验证

本章在从 Github 所爬取的数据集上进行了实验，并选择了本章所提出的 TreeGen 网络作为基础模型。同时，本章将迭代筛选算法与各种命名约束编码策略进行了比较。

3.2.1　实验设置

模型　本实验使用本文第二章中所提出的 TreeGen 模型作为基础模型实现程序生成任务。该模型使用了 5 个自然语言阅读器模块，9 个抽象语法树阅读器模块和 2 个解码器模块，通过 dropout 技术以 0.15 概率进行正则化。对于该模型中使用的所有层的隐含层大小，本章将其设置为 128 维。对于训练，本章使用 Adam[80]优化器在单个 Titan RTX 上以 10^{-4} 的学习率进行模型训练。

对比方法　本章讲所提出的迭代筛选算法与两种方法进行对比。特别的，由于本章的迭代筛选算法关注于标识符命名约束的性质而非标识符命名本身，因而不与标识符命名生成的方法作比较。

原始命名

原始命名直接使用了数据集中所自带的变量、函数名进行训练和生成，原始命名方法在之前的工作中被广泛应用[12,54]。

随机生成算法

随机生成算法是上文中所介绍的算法，该方法在训练的过程中随机对变量、函数名在不影响语义的情况下进行交换。该方法也是迭代筛选算法的一个特例，即当 $K = 1$ 时，迭代筛选算法会退化为该方法。

数据集　本章所使用的数据集从 Github 上的 Java 数据中进行爬取。对于爬取的数据，本章利用其中的注释作为输入的自然语言。为了化简实验难度，本章只选择了程序抽象语法树大小小于 200 的程序作为数据，同时，本章也对数据中的变量命名进行简化，以减少神经网络所需建立的词表大小。在简化的基础上，为了尽可能还原低变量、函数名命名质量的编码难度，本实验通过人工的方法对变量、函数名进行了进一步修改。对于训练集、验证集和测试集，本章随机分配了 20,000、1,000、1,000 条数据。

评价指标　本章的评价指标使用了生成准确率。如果所生成的程序和参考程序具有相同的语义（即在满足可以任意修改变量、函数名的情况下具有完全相同的程序元素），本章则认为其生成正确，否则本章则认为其生成错误。

3.2.2　实验结果

表 3.1 展示了生成程序的实验结果。

在该实验结果中，可以发现直接使用原始命名具有较低的生成准确率（11.2%），该效果符合本章的预期，因为该方法受制于变量、函数名命名风格的影响。随机生成算法通过引入随机交换缓解了该问题，并且取得了更好的效果（18.1%），然而，由于随

表 3.1　实验结果。“迭代筛选算法 -5” 表示 $K = 5$，随机采样 5 个不同的交换进行训练。

行 #	TreeGen	准确率
1	原始命名	11.2%
2	随机生成算法	18.1%
3	迭代筛选算法 -5	**19.1**%

机命名风格的问题，其生成准确率仍旧受到一定影响。迭代筛选算法解决了以上两种问题，并取得了最佳的实验结果（19.1%）。相比于现有方法，迭代筛选算法取得了 8 个百分点的提升。

3.3　小结

本章从程序生成中语义约束感知问题出发，研究了在程序中自身关于变量、函数命名约束感知问题。本章分析了现有程序生成模型的局限性，表明当数据中变量、函数命名质量较差时程序生成的效果会受到影响。基于本章的分析，本章先提出了一个随机生成算法。然而，随机生成算法也受到了随机命名风格的影响。因而，本章进一步提出使用迭代筛选算法对多个命名风格进行采样并使用最佳的命名风格进行训练。本章在对 Github 程序生成任务上进行了实验验证，其结果证明本章所提出的迭代筛选算法的有效性。

第四章　基于集成学习的扰动约束感知

为了解决如第 1.2.4节所分析的扰动约束（对输入自然语言进行不影响程序语法的词语替换后，输出程序的结构应该保持不变，即约束一致性）感知问题，本章提出了一种感知上下文的检测和修复算法（Context-Aware Testing and repair for code generation，CAT）。对于一个程序生成模型来说，其输入是一段自然语言描述，输出是该输入自然语言描述对应的程序。CAT 先从该程序生成模型中自动检测出不满足扰动约束的输入输出（本章称之为约束不一致性问题）。为了实现自动检测，CAT 先对输入的自然语言描述进行变换。对于一条输入句子（自然语言描述），CAT 对该句子中的词语进行相近词语替换以生成替换后的变异句子。在产生变异句子之后，CAT 将其和原本变异前的句子输入到程序生成模型当中，得到其对应的程序。CAT 进一步将变异句子所得程序与原始输入句子所得程序进行比较以验证是否满足扰动约束。当两个程序中理应不发生改变部分的变化高于本章的预定义阈值时，CAT 会报告其为一对不满足扰动约束的输入输出，即约束不一致性问题。

针对所检测出的约束不一致性问题，CAT 的下一步任务则是让该问题中的一对输入输出感知扰动约束。一种合理的解决方案便是对其输出进行修改，使其满足扰动约束。因而，CAT 通过集成学习的方法集成程序生成系统对于多个相似输入的输出的结果，以纯黑盒方式修复约束不一致性问题进而提升程序生成系统所生成程序的可接受性。除此之外，CAT 也提供了一种灰盒修复技术以灰盒的方式修复程序生成系统的约束不一致性问题。

4.1　基于检测和修复的扰动约束感知方法

本节将介绍所提出 CAT 方法的梗概和其详细的步骤。

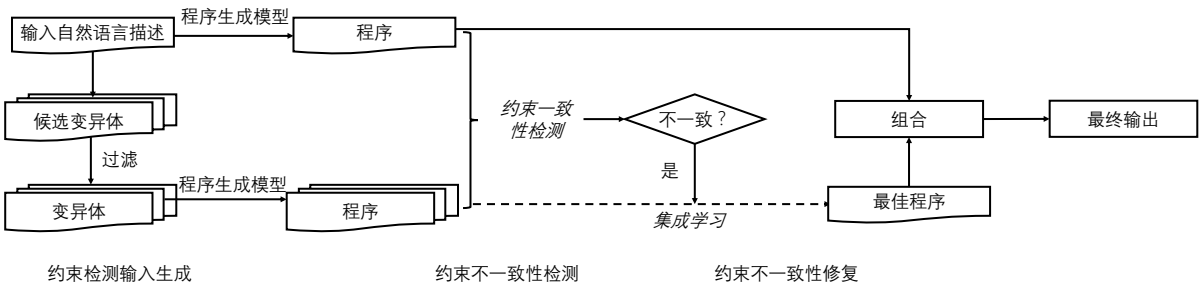

图 4.1　CAT 方法检测和修复程序生成系统的流程

4.1.1 方法梗概

CAT 的大致流程如图 4.1 所示。CAT 基于以下三个主要步骤自动检测和修复程序生成的约束不一致性问题：

1) 约束检测输入生成 该步骤的目标是生成转换后的句子（检测输入）以用于约束一致性检测。对于每个输入句子，CAT 通过上下文相似的单词替换来进行句子变异生成候选变异体，进而使用语义验证来过滤所生成候选变异体。然后将通过语义验证的变异体作为被测程序生成系统的最终检测输入。其详细信息在第 4.1.2节中介绍。

2) 约束不一致性检测 该步骤的目标是引入自动化的不一致性检测。自然语言输入和程序输出之间的扰动约束可以被表示为：原始句子对应的程序及其上下文相似的变异体句子对应的程序在变异词未影响的程序部分的应该具有一定程度上的一致性。为了检测该扰动约束，该步骤使用了相似度度量的方法来衡量程序之间的一致性程度并将其作为检测结果。此步骤的详细信息在第 4.1.3节中介绍。本章也探索了四个不同的相似度衡量指标，这些指标在第 4.2.1.2节中进行了详细描述。

3) 约束不一致性修复 该步骤的目标是自动修复约束不一致性问题。CAT 应用黑盒和灰盒方法根据多个变异体中的最佳程序来转换为原始句子对应的程序。本章探索了两种选择最佳程序的方法，一种使用程序生成系统预测概率，另一种则是使用交叉引用的方法。此步骤的详细信息将在第 4.1.4节中给出。

4.1.2 约束检测输入生成

为了自动生成约束检测的输入，本节提出了两种不同的方法：1）一种轻量级的相似词语替换方法（其对应的测试和修复方法记为 CAT-L）；2）一种基于神经网络的同位词语替换方法（需要依赖于高性能计算，其对应的测试和修复方法记为 CAT-N）。

4.1.2.1 轻量级的相似词语替换方法

本章将该词语替换方法记为 CAT-L。CAT-L 的检测输入生成包括以下两个步骤。

语义相似的替换语料构建 要进行语义相似词替换，其关键步骤是找到一个可以替换为其他（相似词）不损害句子结构及含义的词。通过单词替换生成的新句子应该产生与原始句子一致的程序。

一个词的词向量通过它们的常见的上下文来表示其含义[81]。为了测量相似度，本章使用从文本语料库训练的词向量。在 CAT-L 中，两个词 w_1 和 w_2 之间的词相似度，用 $\mathrm{sim}(w_1, w_2)$ 表示。其由下面的公式计算，其中 $\boldsymbol{v}_x$ 表示单词 x 的词向量。

$$\mathrm{sim}(w_1, w_2) = \frac{\boldsymbol{v}_{w_1}\boldsymbol{v}_{w_2}}{|\boldsymbol{v}_{w_1}||\boldsymbol{v}_{w_2}|} \tag{4.1}$$

为了构建可靠的语义相似词语集，CAT-L 采用两个词向量模型并使用它们结果的交集作为语义相似词语集。第一个模型是 GloVe[81]，它是从 Wikipedia 2014 数据[82] 和 GigaWord 5 数据[83] 训练而来的。第二个模型是 SpaCy[84]，它是一个在 OntoNotes[85] 上训练的多任务卷积神经网络模型，其中训练数据包括从电话对话、新闻专线、新闻组、广播新闻、广播对话和博客中收集的数据。当两个词的相似度在这两个模型中都超过 0.9 时，CAT-L 则认为这两个词是语义相似的，并将其放入语义相似词语集中。CAT-L 使用这种方法总共收集了 131,933 个词对。

输入句子变异　下面本章分别介绍词语替换和句子结构过滤。

1）词语替换：对于原始输入句子中的每个单词，CAT-L 遍历语义相似词语集以确定是否存在匹配项。如果找到匹配项，CAT-L 会将单词替换为其语义相似的单词，并生成一个新的变异输入句子。与原始句子相比，每个变异句子都包含一个替换词。为了减少产生含义不相近的变异体的可能性，该方法在替换的过程中只替换名词、形容词和数词。

2）句子结构过滤：生成的变异句子可能会产生与原来句子不同的含义，因为替换的单词可能不适合变异句子的上下文。例如，“one” 和 “another” 是相似的词，但是 “a good one” 和 “a good another” 并不具有相近的含义。为了过滤这种类型的变异句子，本章提出应用额外的约束来检查生成的变异体。在这里，CAT-L 应用了基于 Stanford 解析器[86] 的结构过滤技术。假设原句为 $s = w_1, w_2, ..., w_i, ..., w_n$, 变异的句子是 $s' = w_1, w_2, ..., w_i', ..., w_n$，其中 s 中的 w_i 替换为 s' 中的 w_i'。对于每个句子，Stanford 解析器输出 $l(w_i)$ 是 Penn Treebank 项目[87] 中的词性标记。如果 $l(w_i) \neq l(w_i')$，CAT-L 则从候选变异体句子中删除 s'，因为该变异会导致句法结构发生变化，从而造成句子含义发生改变。

4.1.2.2　基于神经网络的同位词语替换方法

基于神经网络的同位词语替换方法（记为 CAT-N）和上述轻量级的方法一样，依旧遵循基于单词替换的方法的一般过程：给定输入句子 s 和 s 中的单词 w，CAT-N 识别一组单词（表示为 W_w），其中每一个单词都可以用来替换 s 中的 w。对于每个单词

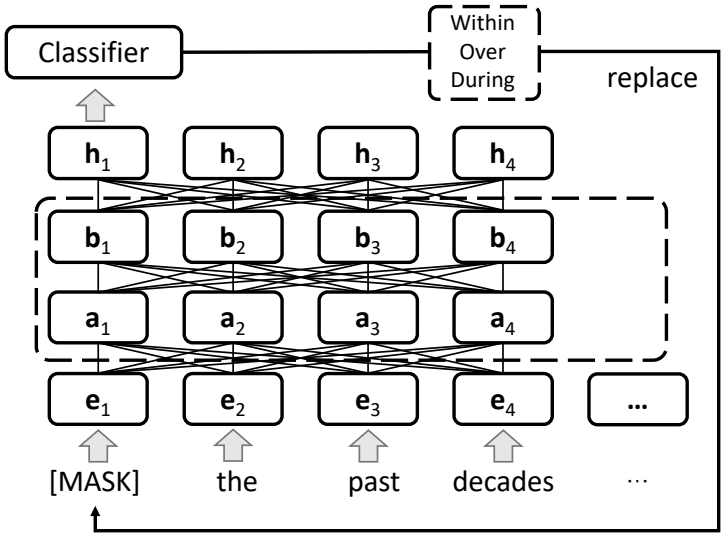

图 4.2　基于 BERT 的上下文感知的词替换

$w_f \in W_w$，CAT-N 进一步确保用 s 中的 w_f 替换 w 是同位替换（替换一个词，并使其对句子含义的变化产生微小的影响），这只会对 s 句子的含义造成微小影响。实现同位替换的关键思想是在 s 的上下文中评估 w 和每个候选替换词 $w_r \in C_w$ 之间的上下文感知的语义相似度。

实现同位替换需要两个阶段。在第一阶段，CAT-N 为输入句子 s 中的每个单词 w 生成一组单词替换候选 C_w。第二阶段，CAT-N 从候选词 C_w 中识别出最终的词集 W_w，通过评估 $w_r \in C_w$ 中的每个单词与句子 s 中的单词 w 之间的上下文感知的语义相似度来实现同位替换。

候选词生成　为了实现同位替换，CAT-N 先为输入原句 s 中的每个词 w 生成一组候选词 C_w，其中 $W_w \subset C_w$。然后，CAT-N 使用词语基于上下文进一步评估上下文感知的语义相似度。

给定一个输入句子 s，CAT-N 通过确保 C_w 中的每个词也适合该句子的上下文以选择候选词 C_w。为了实现该想法，CAT-N 使用 Transformers (BERT)[88] 的双向编码器表示来编码句子上下文并找出哪些单词适合上下文。

图 4.2 显示了 CAT-N 中上下文感知的单词替换模型。作为基于 Transformer[62] 的预训练模型，对于每一条输入句子 BERT 根据上下文为句子中的每个单词输出一个实值向量。在进行单词替换时，输入句子中待替换的单词先被特殊标记 “[MASK]” 所替换。例如，在图 4.2 中，输入句子 “Within the past things...” 中的单词 “Within” 被 “[MASK]” 所替换。然后将带有 “[MASK]” 单词的句子输入到 BERT 模型，该模型输出一组实值向量。为了进一步实现单词替换，BERT 将 “[MASK]” 这个词的向量输入到一个预先

训练好的线性分类器当中，得到一组具有不同预测概率的替换词列表。

给定一个带有单词序列 $w_1, w_2, \cdots, w_N$，其中 N 是单词的总数，CAT-N 依次使用“[MASK]”对句子中的单词进行替换（每次替换一个单词）并将每个含有“[MASK]”的句子输入到 BERT。

BERT 的输出是一组向量 $\boldsymbol{h}_1\boldsymbol{h}_2\cdots\boldsymbol{h}_N$，它们表示输入词的上下文感知向量表示。然后，一个预训练的线性分类器以“[MASK]”词 $\boldsymbol{h}_{\text{mask}}$ 的向量为输入，输出一组初始候选词 C_i。其中的 C_i 中每个候选词都有一个预测概率。

在该预测中，所有候选词的概率之和为 1.0。然后，CAT-N 使用概率过滤器去除具有较低预测概率的候选词（本章将概率阈值设置为 0.05 以去除最不合适的候选词）。此外，如果该词与 CAT-N 所替换的原始词相同，该词也会被去除。最后，CAT-N 将剩余的单词作为替换词 w_{mask} 的候选词 $C_{w_{mask}}$，其中每个词都可以用来填充输入句子中的“[MASK]”词进行单词替换。

请注意，上述单词生成过程是一种上下文感知的过程，因为 BERT 可以准确地根据句子的剩余部分含义来预测“[MASK]”词的合适形态。

语义验证　同位替换旨在为输入句子 s 中的每个单词 w 识别一组单词 W_w，使得 W_w 中的每个单词仅与 s 上下文中的 w 略有不同。在候选词生成中，CAT-N 为 s 中的词 w 生成一组候选词 C_w。然而，在某些情况下，BERT 会预测一个与原始单词具有相似上下文但含义不同的单词。因此，CAT-N 接下来计算一种上下文感知的语义相似度，并用它来去除 C_w 中不合适的词，以识别最终的词集 W_w。

为了实现 CAT-N 的目的，CAT-N 使用了一种新的基于神经网络的机制来评估上下文感知语义相似性。计算上下文感知语义相似度的关键步骤是测量原始词与其替换词之间的词向量相似度。CAT-N 再次使用 BERT 来获得句中词对应词向量。BERT 是自然语言处理中使用最广泛的 word2vec 方法[88-90]。与上一节中在候选词生成中使用 BERT 不同，CAT-N 不给 BERT 提供含有“[MASK]”的句子，而是直接将整个句子作为其输入。这样，BERT 先将每一个词表示为从训练语料库训练出来的向量。该向量表示该单个词的语义。BERT 进一步考虑句子上下文并计算上下文感知语义向量表示。这样，CAT-N 就得到了句子中每个词的上下文感知的词向量，作为最终的词向量。

然后，CAT-N 计算 s 中的单词 w（记为 $\boldsymbol{h}_a$）和 s_r 中的候选词 $w_r \in C_w$ 之间的上下文感知的语义相似度。通过将 s 中的 w 替换为变异句子中的 w_r（记为 $\boldsymbol{h}_b$），CAT-N 进一步使用他们的词向量的余弦相似度 CosSim 进行语义相似度验证：

$$\text{CosSim}(\boldsymbol{h}_a, \boldsymbol{h}_b) = \frac{\boldsymbol{h}_a\boldsymbol{h}_b}{|\boldsymbol{h}_a||\boldsymbol{h}_b|}. \tag{4.2}$$

CAT-N 的语义验证的详细过程由算法 1 展示。对于一个输入句子 s，s 中的一个词 w，以及通过候选词生成生成的候选词集 C_w，CAT-N 的目标是得到一个最终的词集 W_w。

CAT-N 先定义一个空集 W_w 用于记录通过验证后的单词（第 1 行）。对于 s 中的单词 w，CAT-N 通过 BERT 进一步提取其词向量 $\boldsymbol{h}_a$（第 2 行）。接下来，对于每个候选词 $w_r \in C_w$，CAT-N 用它来替换 s 中的 w 并生成一个变异句子 s_r（第 3 和第 4 行）。对于 s_r 中的词 w_r，CAT-N 提取其词向量 $\boldsymbol{h}_b$（第 5 行）。然后，为了捕捉上述两个词的上下文感知语义相似度，CAT-N 计算它们的词向量 $\boldsymbol{h}_a$ 和 $\boldsymbol{h}_b$ 的余弦相似度 $CosSim$（第 6 行）。如果相似度高于预定义的阈值 $SThreshold$（本章将阈值设置为 0.85），则认为其通过了语义验证（第 7 行）。因此，CAT-N 将单词 w_r 添加到集合 W_w 作为最终生成的单词替换（第 8 行）。在 CAT-N 自动验证完所有单词之后，CAT-N 为 s 中的单词 w 输出最终的替换单词集 W_w（第 11 行）。此过程可以尽可能确保在 s 中用 $w_f \in W_w$ 替换 w 是同位替换。

最后，CAT-N 依次使用单词 $w_f \in W_w$ 替换 s 中的单词 w，并对 s 中的每个单词重复此过程以生成最终的句子集 M_f。CAT-N 将这些最终生成的句子中的每一个都称为变异体[91,92]。这组生成的变异体进一步用于自动检测（每个变异体与原始句子配对作为一个检测输入对）和修复。

Algorithm 3: 语义验证的计算过程

Data: s: 输入句子；w 输入句子 s 中的词语；C_w 词语 w 对应的候选词集合（候选词生成方法的输出结果）
Result: W_w: 最后用以同位替换句子 s 中词语 w 的词集合

```
1  W_w = {}
2  h_a = BERT(s, w)
3  for 每个候选词 w_r ∈ C_w do
4  |   s_r = Replace(s, w, w_r)
5  |   h_b = BERT(s_r, w_r)
6  |   Similarity = CosSim(h_a, h_b)
7  |   if Similarity >= SThreshold then
8  |   |   W_w = W_w ∪ w_r
9  |   end
10 end
11 return W_w
```

CAT-L 和 CAT-N 共用一套测试和修复方法 CAT。为了便于表示，在本章后续方法部分统一使用 CAT 来进行介绍。同时，在本章后续实验中，将使用轻量级的相似词语替换方法的测试和修复方法统称为 CAT-L，将使用基于神经网络的同位词语替换方法统称为 CAT-N。

4.1.3　约束不一致性检测

为了执行约束不一致性检测，CAT 需要对其生成的输入进行验证，即检查是否发现约束不一致性问题。CAT 假设输入句子中的未更改部分保有了它们应有的语义充分性。含义充分性是指生成的程序是否传达了相同的意思，是否有信息丢失、添加或扭曲。

设 $t(s)$ 和 $t(s')$ 是句子 s 和 s' 的对应程序，其中 s' 句子是由 s 句子通过上述单词替换方法替换单词 w 为 w' 得到。为了实现想要的约束不一致性检测，CAT 需要在忽略所替换单词 w 和 w' 对输出程序造成影响的情况下检查两个程序之间的相似性。然而，要自动化忽略单词 w 和 w' 对输出程序造成的影响并不容易。因为 1）w 和 w' 替换可能会改变整个程序，2）不容易准确映射出输入文本中的单词 w 和 w' 所对应的程序元素。

为了绕过这个问题，CAT 通过计算 $t(s)$ 和 $t(s')$ 中子句（程序序列的子序列）的相似度并利用多个子句相似度的最大值来近似估计 $t(s)$ 和 $t(s')$ 之间未受影响部分的一致性水平。算法 2 展示了这个过程。对于 $t(s)$ 和 $t(s')$，CAT 先应用 GNU Wdiff[93] 来获得不同程序之间的差异切片（第 1 行）。GNU Wdiff 以词为基础比较程序，该方法对于比较只有少量修改的两个相似程序很有效[93]。使用 Wdiff，两个程序 “**A** B C **D** F” 和 “**B** B C **G H** F" 之间的差异切片分别表示为前一个程序中的 “A” 和 “D” 以及后一个程序中的 “B” 和 “G H’’。

CAT 将 $t(s)$ 和 $t(s')$ 的差异切片分别保存到集合 B_s 和 $B_{s'}$ 之中。然后 CAT 依次从程序中删除一个差异切片的差异部分，每次只删除一个（第 5 行和第 9 行）。通过这样的方式，CAT 会得到一个程序的多个子序列。对于上面的例子，“**A** B C **D** F’’ 将有两个子序列：“B C D F’’（删除 “ A’’）和 “A B C F”（删除 “D”）。CAT 将 $t(s)$/$t(s')$ 的这些新的子序列添加到集合 T_o/T_m 中（第 6 行和第 10 行）。

对于集合 T_o 中的每个子序列元素，CAT 将其与集合 T_m 中的每个子序列元素两两之间计算它的相似性①（第 15 行）。因此，CAT 得到 $|T_o| * |T_m|$ 个不同相似度分数，其中 $|T_o|$ 和 $|T_m|$ 分别是集合 T_o 和集合 T_m 的大小。然后，CAT 使用其中最高的相似度值作为最终的一致性得分的结果（第 16 行）。

这种策略减少了变异词对程序的影响，并有助于找出不一致性分数的上限。即使相似度最大的两个子序列包含所替换的词对应的程序修改部分，程序其他部分的相似度也比之会更差，因此这种情况不太可能因为含有替换词而导致产生偏差，从而导致误报。

① 本章探讨了四种类型的相似性度量（其详细信息参见章节 4.2.1.2 中）。

Algorithm 4: 一致性分数计算过程

Data: $t(s)$: 原始输入句子对应的程序；$t(s')$: 变异体对应的程序
Result: ConScore: $t(s)$ 和 $t(s')$ 之间的一致性分数

```
1  B_s, B_s' = Wdiff(t(s), t(s'))
2  T_o = {t(s)}
3  T_m = {t(s')}
4  for 每个子序列 b_s ∈ B_s do
5  |   r = DeleteSub(t(s), b_s)
6  |   T_o = T_o ∪ {r}
7  end
8  for 每个子序列 b_s' ∈ B_s' do
9  |   r' = DeleteSub(t(s'), b_s')
10 |   T_m = T_m ∪ {r'}
11 end
12 ConScore = -1
13 for 每个序列 a ∈ T_o do
14 |   for 每个序列 b ∈ T_m do
15 |   |   Sim = ComputeSimilarity(a, b)
16 |   |   ConScore = Max(ConScore, Sim)
17 |   end
18 end
19 return ConScore
```

4.1.4 约束不一致性修复

本节先介绍约束不一致性修复过程的整体情况，然后介绍两种不同的修复过程中程序排序策略（基于概率的和基于交叉引用的）。

4.1.4.1 整体的修复过程

CAT 先修复原始句子对应的程序，然后修复变异体对应的程序，使得其通过 CAT 的程序一致性测试。

算法 3 显示了整体的修复过程。对于已发现存在约束不一致性问题的程序 $t(s)$，CAT 生成一组变异体并得到它们的程序 $t(s_1), t(s_2), ..., t(s_n)$。这些变异体及其程序，连同原始句子及其程序，被放入字典 T 中（第 1 行）。然后，CAT 使用预测概率或交叉引用的方式按降序排列 T 中的元素，并将结果放入 $OrderedList$（第 2 行）。第 4.1.4.2节和第 4.1.4.3节给出了预测概率和交叉引用排名的详细信息。

接下来，CAT 应用词对齐来获得 s 和 $t(s)$ 之间的映射词并记为 $a(s)$（第 3 行）。词对齐是一种自然语言处理技术，当且仅当两边的词具有程序关系时，它才将两个词连接起来。特别的，CAT 采用了基于全匹配的词对齐技术。在词对齐之后，CAT 检查是否可以采用 $OrderedList$ 中的句子对 $(s_r, t(s_r))$ 来修复原始程序。CAT 按照排名顺序，

Algorithm 5: 自动化修复过程

Data: s: 一个输入语句; $t(s)$: s 在程序生成系统中对应的程序; $s_1, s_2, ..., s_n$: s 通过自动输入生成所生成的变异体; $t(s_1), t(s_2), ..., t(s_n)$: 变异体所得程序;
Result: NewTrans: s 程序的修复结果

$T = \{(s, t(s)), (s_1, t(s_1)), (s_2, t(s_2)), ..., (s_n, t(s_n))\}$
OrderedList = Rank(T)
$a(s)$ = wordAlignment($s, t(s)$)
NewTrans = $t(s)$
for 每条输入语句及其对应的输出程序 $s_r, t(s_r) \in$ OrderedList **do**
    if $s_r == s$ **then**
        break
    end
    $a(s_r)$ = wordAlignment($s_r, t(s_r)$)
    w_i, w_i^r = getReplacedWord(s, s_r)
    $t(w_i)$ = getTranslatedWord($w_i, a(s)$)
    $t(w_i^r)$ = getTranslatedWord($w_i^r, a(s_r)$)
    $t^r(s_r)$ = mapBack(t(s), t(s_r), s, s_r, a(s), a(s_r))
    if *isinaligner(w_i^r))* **then**
        NewTrans = $t^r(s_r)$
        $s_o, t(s_o)$ = getRepairedResult(s)
        if *not* $isConsistent(t(s_o), t^r(s_r))$ **then**
            continue
        end
    end
    if isTest(s) **then**
        $s_o, t(s_o)$ = getRepairedResult(s)
        if *not* $isConsistent(t(s_o), t^r(s_r))$ **then**
            continue
        end
    end
    NewTrans = $t^r(s_r)$
    break
end
return NewTrans

直到找到一种可以满足约束不一致修复要求的变异程序。

如果 s_r 是原句（$s_r == s$），这意味着原句对应的程序被认为是比其他变异体对应的程序更好的选择，所以 CAT 不会对其程序进行任何修改（第 6-8 行）。否则，CAT 对 s_1 和 $t(s_1)$ 进行与 s 和 $t(s)$ 相同的对齐。变量 w_i, w_i^r 表示 s, s_r 中的替换词，CAT 通过对齐得到其对应的程序部分 $t(w_i), t(w_i^r)$（第 9-12 行）。

由于词对齐不是百分之百准确的，同时由于程序的特殊性，大量相似的输入理应生成相同的程序。因此，CAT 使用了以下规则来完成程序映射：如果所替换的词并没有存在于词对齐列表中，我们默认其生成的程序不需要任何修改（第 13-15 行）。

在修复变异体的程序时（第 16 行），CAT 已知原句的修复结果（第 17 行），然后检查候选程序方案是否与修复后的原句对应程序一致（第 18-20 行）。如果不一致，CAT 继续尝试选择其他候选者。

4.1.4.2 基于概率的程序排序

对于一个句子 s 及其变体 $S = s_1, s_2, ...s_n$, $t(s)$ 表示为 s 对应的程序，$t(s_i)$ 表示为变异体 s_i 对应的程序。该排序方法记录每个 $t(s_i)$ 的程序生成概率，并选择概率最高的变异体作为程序映射候选者。然后将相应变异体的程序映射回待修复程序，以使用单词对齐生成 s 对应的最终程序。

这是一种灰盒修复方法。它既不需要训练数据也不需要训练算法的源代码，但需要程序生成系统提供的预测概率。本章将此称为灰盒修复方法，因为实现者可能将此通常不能访问到的概率信息视为该方法的内部属性。

4.1.4.3 基于交叉引用的程序排序

对于一个句子 s 及其变体 $S = s_1, s_2, ...s_n$, $t(s)$ 表示为 s 对应的程序，$t(s_i)$ 表示为变异体 s_i 对应的程序。该方法计算 $t(s), t(s_1), t(s_2), ...t(s_n)$ 之间的相似度，并使用与其他程序最近（具有最大平均相似度分数）的程序映射回并修复原始输入对应的程序。

这是一种黑盒修复方法。它只需要执行被测程序生成系统和该系统的程序输出。

4.2 实验验证

4.2.1 实验设置

本节将介绍验证自动检测输入生成、自动不一致性检测和自动不一致性修复的实验设置。

4.2.1.1 研究问题

本章通过评估 CAT 生成可用于约束一致性检测检测输入的能力来开始本章的研究。因此，本实验首先关注：

RQ1：CAT 方法生成检测输入的准确率如何？ 本章通过随机抽样一些生成的检测输入对并（人工）检查它们是否有效来回答这个研究问题。这个问题的答案确保 CAT 确实生成了适合约束一致性检测及修复的输入。

鉴于本问题发现了 CAT 可以生成有效检测输入对的证据，本章进一步将注意力转向 CAT 在检测约束一致性问题方面的有效性问题。因此本章提出了第二个研究问题：

RQ2：CAT 揭示约束不一致性问题的能力如何？ 为了回答 RQ2，本章根据相似度指标计算一致性分数以作为约束不一致性检测标准（用来检测是否存在一致性问题）。为了评估 CAT 的约束一致性问题揭示能力，本章通过抽样的方式手动检查检测结果，并将手动检查结果与自动检测结果进行比较。

在研究了该问题之后，本章进一步评估了 CAT 的修复步骤，以了解它修复约束不一致性问题的能力。因此，本章提出：

RQ3：CAT 对约束不一致性问题的修复能力如何？ 为了回答这个问题，本章评估了 CAT 修复约束不一致性问题的数量及比例（通过一致性指标和手动检查进行评估）。同时，本章还人工检查了由 CAT 修复的程序，并检查它们是否提高了程序的一致性和可接受性。

4.2.1.2 相似度衡量指标

本章探索了四种广泛采用的相似性指标来衡量生成程序的相似度。为了便于说明，本章用 t_1 来表示原始输入的对应的程序；本章使用 t_2 来表示变异体输入的对应的程序。

基于 LCS 的相似度衡量指标 它通过 t_1 和 t_2 之间最长公共子序列的长度来衡量相似度：

$$M_{LCS} = \frac{len(LCS(t_1, t_2))}{Max(len(t_1), len(t_2))} \tag{4.3}$$

在这个公式中，LCS 是一个计算在 t_1 和 t_2 之间以相同顺序出现的最长公共子序列[94] 的函数。例如，输入序列 “A B C D G H” 和 “A E D F H R” 的 LCS 是长度为 3 的 “A D H”。$len(t_1)$ 和 $len(t_2)$ 则表示的是序列 t_1 和 t_2 的长度值。

基于 ED 的相似度衡量指标 这个指标的计算基于 t_1 和 t_2 之间的编辑距离。编辑距离是一种通过计算将一个字符串转换为另一个字符串所需的最小操作数来量化两个字符串的不同程度的方法[95]。为了归一化编辑距离，本章使用了之前工作中也采用的以下公式[96,97]。

$$M_{ED} = 1 - \frac{ED(t_1, t_2)}{Max(len(t_1), len(t_2))} \tag{4.4}$$

在这个公式中，ED 是一个计算 t_1 和 t_2 之间编辑距离的函数。

基于 tf-idf 的相似度衡量指标 tf-idf（term frequency–inverse document frequency）可用于衡量词频方面的相似性。每个单词 w 都有一个权重因子，根据以下公式计算，其中 C 是文本语料库，$|C|$ 是句子数，f_w 是 C 包含 w 的句子的数量。

$$w_{idf} = \log((|C| + 1)/(f_w + 1)) \tag{4.5}$$

本章用词袋模型来表示每个程序[98]，这是自然语言处理中常用的一种简化表示。在这个模型中，语法和顺序被忽略，只考虑程序元素出现的频次（即 “A B C A” 表示为 “A”: 2，“B”: 1，“C” :1，即向量中的 [2, 1, 1]）。向量的每个维度都乘以它的权重 w_{idf}。本章计算 t_1 和 t_2 的加权向量的余弦相似度 (公式 4.1) 作为它们最终的基于 tf-idf 的一致性分数。

基于 BLEU 的相似度衡量指标 BLEU（Bi-Lingual Evaluation Understudy）是一种通过检查机器输出结果与人类给出的参考结果之间的对应关系来自动评估程序生成质量的算法。它也可以用来计算原句对应程序和变异体对应程序之间的相似度。BLEU 分数的详细信息、描述和动机可以在程序文献中找到[99]。由于篇幅有限，这里只做一个概述。

BLEU 先统计句子（在本章中是程序）和句子之间匹配子序列的数量计算精度 p_n（称为修改后的 n-gram 精度[99]，其中 n 表示子序列长度）。例如，在句子 “A A B C” (s_1) 和 “A B B C” (s_2) 中，s_2 中有三个 2-gram 子序列：$A\ B$、$B\ B$ 和 $B\ C$。其中两个与 s_1 中的匹配：$A\ B$ 和 $B\ C$。因此，p_2 是 2/3。

与 p_n 一样，BLEU 分数的计算还需要指数惩罚因子 BP 以惩罚过短的程序，如公式 4.6 所示。c 表示 $t(s_i)$ 的长度，r 是 $t(s)$ 的长度。

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases} \tag{4.6}$$

BLEU 分数最后由公式 4.7计算得到，在此公式中，权重 $w_n = \frac{1}{N}$ (本章设置 $N = 4$)。

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right). \tag{4.7}$$

由于 BLEU 是单向的（即 $BLEU(s, s') \neq BLEU(s', s)$），因而本章使用其两种分数中较高的分数来衡量 s 和 s' 之间的相似性。这与本章在公式 4 中的意图是一致的：计算获得一致性度量的上限，从而避免关于因为程序问题而导致的误报。

4.2.1.3　程序生成系统

本章的实验考虑了常用的 CodeGPT 程序生成系统。本章使用 CodeGPT，因为它是一个在程序生成中常见的开源模型，也是一个高质量的程序生成系统。

本章将 CodeGPT 迁移到本章所使用的数据集。本章使用默认设置来训练 CodeGPT。CodeGPT 基于一个程序生成数据集（Concode 数据集）进行训练。Concode 数据集[100]包含 100,000 条训练数据，其中每条数据包括其自然语言及上下文代码。验证数据（帮助调整超参数）也来自 Concode 数据集，其中包含 2,000 条输入自然语言描述。我们使用 Transformers 与 Pytorch 深度学习库对模型进行训练[101]。

4.2.1.4　测试集

本章使用来自 Concode 的验证集[100] 数据集作为测试集。该数据集用以检测 CodeGPT 模型的稳定性。验证集包含不同于训练集中的 2,000 条数据。本章从中随机选择了 500 条数据作为最终的测试集。

4.2.1.5　实验参数设置

CAT-N 的实现基于公开可用的标准 BERT[101]。BERT 模型包含 24 层、1024 大小的隐含层、16 个头。该模型在 16 个 TPU 芯片上进行了 100 万步的训练，训练的批次大小为 256。

本章在检测过程中为每个输入句子生成最多 5 个有效变异体，在修复过程中为每个输入句子中生成最多 16 个有效变异体。

本章的实验是在 256GB RAM 和四个总共包含 32 个内核的 Intel E5-2620 v4 CPU（2.10 GHz）的 Ubuntu 16.04 上进行的。本章使用的神经网络都是在八块 Nvidia Titan RTX（24 GB 内存）上训练和预测的。

4.2.2　实验结果

本节回答研究问题对应实验结果。

4.2.2.1 对输入生成的效果 (RQ1)

要回答这个问题，对于本章数据集中的每条输入句子（总共 500 个），本章分别使用检测和修复算法 CAT-N 和 CAT-L 的方法生成变异体。每个变异体与原始句子配对以形成检测输入。然后记录数量（即检测输入的总数）及有效性（即检测输入中的变异和原始句子是否合适于用作约束不一致性检测）。

数量： 对于 501 个输入句子，CAT-N 通过同位替换总共生成了 2,362 个变异体，这些变异体可以与原始句子配对，作为程序生成模型的检测输入。对于 CAT-L，它则产生 163 个检测输入。

有效性： 有效性衡量生成的检测输入句子符合约束不一致性要求的比例。如果生成的变异体 1) 语法正确；2）语义上合理；3）应该产生与原始句子语义相同的程序（不包含所替换的单词所对应的程序部分），本章则认为检测输入是有效的。本章为 CAT-N 和 CAT-L 分别随机抽取 50 个输入用例。

本章的人工检查表明 CAT-N 生成的检测输入中有 82% 是有效的。由于数据集是从代码注释中收集的，因而其中存在很多不确定性描述，为检测输入生成带来了困难。如果输入描述越接近于常见自然语言，其对应的输入生成的有效率越高。对于 CAT-L，其检测输入的 74% 是有效的。这些结果表明 CAT-N 相比于轻量级的 CAT-L 不仅生成了更多的检测输入，而且还生成了更多合格的检测输入，用于检测约束不一致性问题。

总的来说，对于 RQ1，本章有以下结论：

> **对 RQ1 的回答**：CAT-L 可以有效生成 163 个检测输入，其生成输入的有效率为 74.0%。CAT-N 可以有效生成 2,362 个检测输入，其生成输入的有效率为 82%。

4.2.2.2 CAT 在揭示约束不一致性问题方面的能力 (RQ2)

表 4.1 所检测出的约束不一致性问题数量 (RQ2)

	指标	CAT-L	CAT-N
CodeGPT	LCS	75 (46%)	928 (39%)
	ED	75 (46%)	928 (39%)
	TFIDF	77 (47%)	926 (39%)
	BLEU	73 (45%)	911 (39%)

在本实验中，本章设定四个相似度度量指标 LCS、ED、tf-idf 和 BLEU 的阈值为 0.963、0.963、0.999、0.906，该阈值根据人工标注的数据通过搜索的方法得到。对于

CAT-N 和 CAT-L 生成的每个检测输入，本章将其输入 CodeGPT 当中以获取其对应程序，然后应用相似度指标及阈值来自动化确定其是否存在约束不一致性问题（低于我们设定的阈值）。

表 4.1 显示了 CAT-N 和 CAT-L 报告的每个相似性指标的对应检测到的问题数量。对于 CAT-L 而言，在 CodeGPT 上不一致的生成程序个数和相比于总输入用例数量的百分比分别是 LCS: 75 (46%)；ED：75 (46%)；tf-idf: 77 (47%)；BLEU：73 (45%)。因此，总体而言，大约五分之二的程序低于本章选择的一致性阈值。本章观察到 CAT-N 在报告的问题数量上明显优于 CAT-L。对于 CAT-N 而言，在 CodeGPT 模型上不一致的程序个数和相比于总输入用例数量的百分比分别是 LCS: 928 (39%)；ED：928 (39%)；tf-idf: 926 (39%)；BLEU：911 (39%)。平均而言，CAT-N 在 CodeGPT 模型上检测到的问题比 CAT-L 多十余倍，但是所占输入用例数量的百分比有所下降。

为了验证检测结果的有效性，本章统一从每种方法的检测结果中采样 50 个报出约束不一致性问题的输入用例。对于每个采样的检测输入及其程序生成结果，本章手动检查检测输入及其输出程序是否存在不一致的问题。

本章展示了所报出问题的准确率的人工检查结果。结果表明，对于 CAT-L，其可以以 72% 的准确率找出 CodeGPT 中所包含的约束不一致性问题。CAT-N 同样以 72% 的准确率找出 CodeGPT 中所包含的约束不一致性问题。这些结果表明 CAT-N 报告的问题的有效性与 CAT-L 相似。总的来说，本章的人工检查表明 CAT-N 报告的问题的有效性类似于 CAT-L 报告的问题。

> 对 **RQ2** 的回答：在检测 CodeGPT 模型中的约束不一致性问题时，CAT-L 可以检测平均 75 个约束不一致性问题，而 CAT-N 则可以多检测到十倍多的约束不一致性问题。CAT-L 的检测准确率为 72%，CAT-N 和 CAT-L 具有类似的效果，其准确率也为 72%。

4.2.2.3　CAT 在修复约束不一致性问题方面的能力 (RQ3)

为了便于比较，本章让 CAT-N 和 CAT-L 修复同一组检测到的问题，即由 CAT-N 检测到的问题。对于每个问题，本章让 CAT-N 和 CAT-L 在原输入句和测试用例中的变异体上都生成最多 16 个变异体（第 4.2.1.5 节介绍的修复变异数上限）进行自动修复。

为了回答 RQ3，本章先展示了通过相似度度量计算得到的已修复问题的数量。最后，本章展示了通过人工检查所生成的程序的修复效果。

通过相似性度量访问的修复效果　结果显示在表 4.2 中。每个单元格根据相似性指标显示已修复的问题数量，以及已修复问题的比例（相对于 CAT 所检测到的问题总数）。

CAT-L 在 CodeGPT 上以黑盒/灰盒修复的形式可以修复平均 4%/4% 的约束不一致性问题。本章观察到 CAT-N 修复的问题比 CAT-L 多。以黑盒修复的方式来说，CAT-N 修复了 42% 的报告问题，而 CAT-L 仅修复了 4%。平均而言，CAT-N 在 CodeGPT 上修复的问题比使用黑盒修复（使用交叉引用）的 CAT-L 多 950%，比在 CodeGPT 上使用灰盒修复的 CAT-L 多 725%（使用预测概率）。

表 4.2　修复约束不一致性问题的数量和比例 (RQ3)

方法	指标	基于预测概率	基于交叉引用
CAT-L	LCS	37 (4%)	38 (4%)
	ED	37 (4%)	38 (4%)
	TFIDF	55 (6%)	58 (6%)
	BLEU	20 (2%)	20 (2%)
CAT-N	LCS	308 (33%)	378 (41%)
	ED	308 (33%)	377 (41%)
	TFIDF	308 (33%)	398 (43%)
	BLEU	302 (33%)	397 (44%)

修复程序的有效性　本章手动检查了 LCS 相似性指标上使用交叉引用的方式对 CodeGPT 模型的约束不一致性问题修复结果。目的是检查基于相似度度量的问题修复是否确实提高了生成程序的一致性。

在验证 CAT 修复生成程序约束不一致性问题的基础上，本章验证了感知扰动约束的最终目标（提高生成程序可接受性的能力）。可接受性表示人工检测对程序生成效果的评价。在很多情况下，即使程序生成没有生成正确的程序，其提供的参考程序也有效的辅助了开发人员的开发。因此，人工检查考虑两个方面：1）修复前后生成程序的一致性；2) 修复前后的程序可接受性。对于每个维度，本章设置了三个标签 “提升”、“不变” 和 “下降”。本章随机抽样了 50 个所报告的修复（由于 CAT-L 的修复数量不够，因而对于 CAT-L 本章只抽样了 38 个修复）。对于可接受性而言，本章检查了原始输入生成的程序和变异体生成的程序的修复，因此需要进行 100（76）次人工检查。

表 4.3 显示了实验结果。从该表中，为了提高一致性，CAT-N 的修复成功地提高了 45 (90%) 个生成程序的一致性，其余 5 (10%) 个生成程序的一致性保持不变。而对于 CAT-L，它的修复提高了 36 (95%) 个生成程序的一致性，1 个 (3%) 生成程序的一致性保持不变，1 个 (3%) 生成程序的一致性有所下降。

为了验证生成程序的可接受性，CAT-N 提高了 30 个 (30%) 生成程序可接受性，降低了 27 (27%) 个生成程序的可接受性。CAT-L 提高了 9 个（24%）生成程序的可接受

性，但导致 6 (16%) 个程序的可接受性下降。特别的，在标记数据的过程中，本章发现该程序可接受性的提升依赖于原程序生成技术的生成效果。如果程序生成技术对一条输入句子及其测试具有效果较好的生成效果，则其可接受性的提升比例会较大，否则，可接受性的提升比例会较小。

表 4.3　关于问题修复的人工检查结果 (RQ3)

	角度	提升	不变	下降
CAT-L	**一致性**	36	1	1
	可接受性	9	61	6
CAT-N	**一致性**	45	5	0
	可接受性	30	43	27

对 **RQ3** 的回答：CAT-L 的黑盒修复平均减少了 4% CodeGPT 的生成程序约束不一致性问题。同样的，其灰盒修复平均减少了 4% 的生成程序约束不一致性问题。人工检查表明，修复后的程序在 95% 的情况下提高了一致性（降低了 3%），在 24% 的情况下具有更好的可接受性（16% 更差）。CAT-N 修复的问题数量是 CAT-L 的十倍。

4.3　小结

本章研究了输入自然语言和输出程序之间应该满足的扰动约束。

本章提出了 CAT，这是第一种自动化检测和修复程序生成系统的方法。CAT 接受一个句子作为输入并应用上下文相似的变异对其进行微小改动，用以检测程序生成系统。约束不一致性检测是通过比较原始句子对应程序和变异句子对应程序来进行的。为了判断是否满足扰动约束（一致性），CAT 计算生成程序所得子序列的相似度。当输入未更改部分对应的程序变化高于阈值时，CAT 认为这是一个约束不一致性问题。进而，CAT 通过参考多个变异句子的程序对原始生成的程序以集成学习的方式自动化修复约束不一致性问题。实验结果表明，CAT 可以修复约束不一致性问题进而提升程序生成技术所生成程序的可接受性。

第五章　迭代筛选算法和检测和修复算法的拓展应用

正如第1.3.4节所分析的，本章主要研究本文所提出的约束感知问题并不局限于程序生成。而针对这些约束感知问题所提出的约束感知方法同样可以拓展到其他领域。

命名约束感知问题本质上是对于一条数据中的名称等元素作替换不会影响最终结果的感知问题。该问题也同样存在于无属性图当中。在该数据结构中，每一个节点都是无属性节点，其对应的节点名称不具有任何实际意义并可以任意修改。因而，本文将迭代筛选算法推广到无属性图分类任务之上（第5.1节）。

扰动约束感知问题本质上对于多条相近数据中应该具有某种关联关系的感知问题。该扰动约束也存在于通用的机器翻译当中，对输入做某种相近变换，其对应的翻译也应需要做出相应的相近变换。因此，本文将 CAT 推广到机器翻译任务之上（第5.2节）。

5.1　基于迭代筛选算法的无属性图分类任务

在图神经网络中，一个常见的场景是没有对应的与节点相关的属性（称为*无属性图*）。例如，大规模匿名社交网络的社区检测任务中可能缺少节点人的身份信息（出于隐私问题[102]）。另一种情况是如程序中变量、函数名命名一样，节点的属性是人工随意标记的，这类属性不具有任何语义上的含义。图 5.1 展示了一个布尔可满足性 (SAT) 求解问题的图[103]。其中 x_1 和 c_1 是 *文字*（变量或其否定变量）和 *子句*（变量的析取），并且其满足可以在不改变公式性质的情况下任意重命名。也就是说，在无属性图中也存在标识符的命名约束。

为了感知这类图中的命名约束，本文将迭代筛选算法推广至该任务。

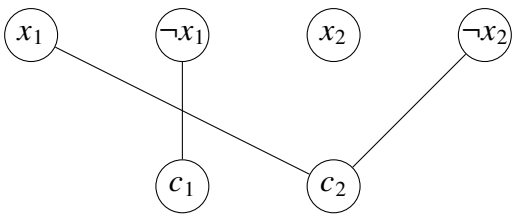

图 5.1　一个 SAT 问题的式子可以被一个图来表示，在这种情况下，一个节点 x_i 代表的是文字（一个变量和其取反的变量），节点 c_i 则表示的是语句（不同文字之间的析取）。一个 SAT 式子则是不同语句之间的合取（如此图中的式子 $\neg x_1 \wedge (x_1 \vee \neg x_2)$）

5.1.1 迭代筛选算法的拓展

本节先对问题进行形式化描述，在此基础上，本节描述了迭代筛选算法的拓展形式。最后，本节进一步分析了迭代筛选算法的在无属性图分类任务上的有效性。

5.1.1.1 问题形式化描述

无属性图上的问题可以形式化表示为在给定图 X 的情况下预测得到对应的输出 Y。本章定义一个特定于任务的谓词函数 $H(X,Y)$ 以表示 Y 是否是给定的输入 X 的输出。在此谓词函数中，当且仅当 Y 是 X 的解时，谓词为真。

对于一个无属性图 $G=\langle V,E\rangle$，其输入可以用一个邻接矩阵 $X\in\{0,1\}^{n\times n}$ 来表示，其中，n 表示的是图中的节点数。在节点分类任务中，其输出是一个矩阵 $Y\in\mathbb{R}^{n\times k}$，表示图上的 n 个节点和其对应的 k 个分类。

为了分析节点信息如何影响（或不影响）图神经网络，本章引入符号 S_n 来表示 $[n]$ 中的变换集合。给定 $\pi\in S_n$，π 在一个无属性图 $X\in\{0,1\}^{n\times n}$ 上的对 X 的动作可以定义为

$$(\pi(X))_{i,j}=X_{(\pi^{-1}(i)),(\pi^{-1}(j))} \tag{5.1}$$

同时，其对 Y 的对应动作为

$$(\pi(Y))_{i,c}=Y_{(\pi^{-1}(i)),c} \tag{5.2}$$

其中，$Y\in\mathbb{R}^{n\times k}$。换句话说，$\pi$ 表示对 X 的行和列以及 Y 中对应的行进行相同的变换。这里，π 是从节点索引到待变换索引的映射，π^{-1} 则相反，其表示从 $\pi(X)$ 和 $\pi(Y)$ 中获得的原本的对应于原始变换前节点的索引映射。

等变性 本节进一步分析图神经网络的等变性节点分类任务问题，该分类需要满足等变性性质（*equivariance property*）。它本质上相当于对于任意排列 $\pi\in S_n$，

$$H(X,Y)\quad\text{implies}\quad H(\pi(X),\pi(Y)) \tag{5.3}$$

也就是说，如果对节点 X 的顺序进行变换，其对应的解 Y 也应该相应地做出改变。

假设对于每个输入图 X 存在一个唯一的满足 $H(X,Y)$ 的输出分类 Y，其从 X 到 Y 的映射可以用函数 h 表示，而其等变性性质变为本章通常所看到的形式：对于每个排列 $\pi\in S_n$，都满足

$$h(\pi(X))=\pi(h(X)) \tag{5.4}$$

在上面的公式中，本章定义了节点分类任务的等变性属性。实际上，该等变性也

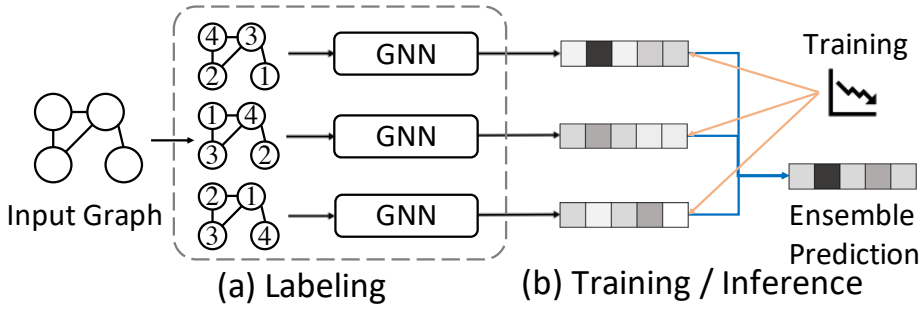

图 5.2　所提出的迭代筛选算法

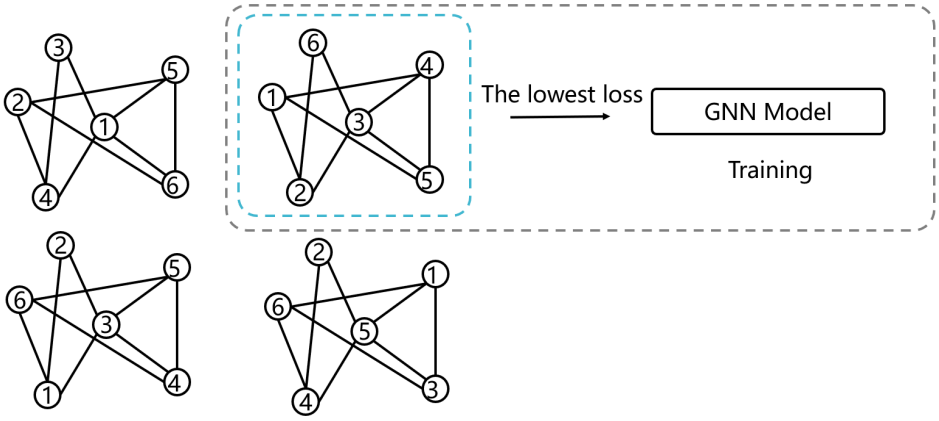

图 5.3　迭代筛选算法的训练过程

可以用图神经网络输出 $f(X)$ 来表示

$$f(\pi(X)) = \pi(f(X)) \tag{5.5}$$

5.1.1.2　迭代筛选算法的具体拓展形式

本节描述了迭代筛选算法的具体拓展形式。

对于训练来说，迭代筛选算法为图上节点随机分配标签。节点由基于分配的标签通过嵌入的方式转变为向量表示。如图 5.3所示，迭代筛选算法在每个训练代中为输入的无属性图动态采样多个标签分配，但图神经网络只训练具有最佳效果的标签（即损失最低的标签）的数据。在下一个训练代中处理图时，迭代筛选算法会重新分配节点标签并进一步重复此训练过程。

从形式上来说，迭代筛选算法在图神经网络模型中分配 $\boldsymbol{e}_1 \cdots \boldsymbol{e}_N$ 作为嵌入参数，其中 N 是嵌入表格的长度。这些嵌入信息没有与任何图或节点相关联。在每个训练代中处理具有节点 $V = \{v_1, \cdots, v_n\}$ 和 $n \leq N$ 的无属性图时，迭代筛选算法随机采样一个变换 $\pi \in S_n$。按照本章的定义，π 对邻接矩阵进行运算；因此，节点 v_i 可以被表示为

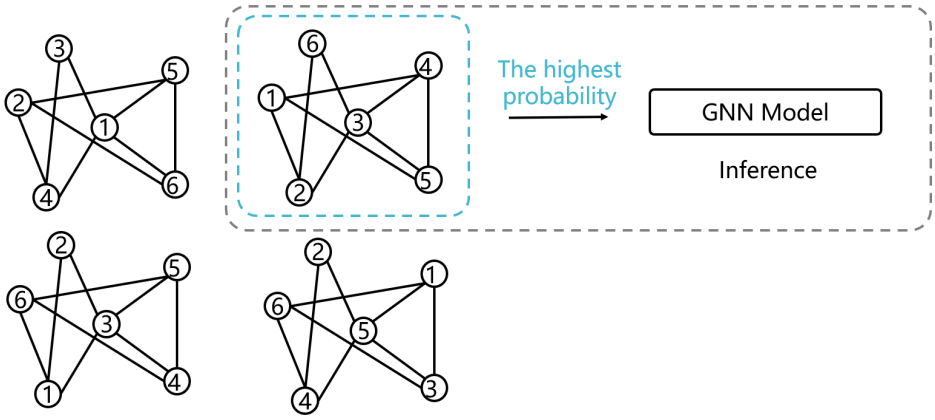

图 5.4　迭代筛选算法的预测过程

$\boldsymbol{e}_{\pi^{-1}(i)}$，用于后续的图神经网络处理。

迭代筛选算法重复这个采样过程 K 次，并计算这些变换在输入到图神经网络中的损失函数值。最后，迭代筛选算法选择损失最小的变换作为用于训练的最终无属性图的标签。在不同的训练代中，即使对于相同的数据样本，这些变换也会被重新采样。

以上的训练过程可以描述为

$$\underset{\omega}{\text{minimize}} \sum_{(X,Y)\in\mathcal{D}} \min_{\pi\in S_n} \sum_{i=1}^{n} D_{\text{KL}}\left((\pi(Y))_i \,||\, f_i(\pi(X);\omega)\right) \tag{5.6}$$

本章用最好的变换 π 训练模型（对 X 和 Y 矩阵的变换，第 5.1.1.1节）。

考虑 k 分类问题的预测过程。对于变换 π_m 的图所对应图神经网络的输出概率为 $\left(p_1^{(m)},\cdots,p_k^{(m)}\right)$。如图 5.4所示，迭代筛选算法选择具有最高联合预测概率（节点概率的乘积）的预测作为其最终的输出。也就是说，迭代筛选算法方法通过公式

$$c = \text{argmax}_i\{p_i\} \tag{5.7}$$

来预测标签。

5.1.1.3　迭代筛选算法在无属性图分类上的有效性分析

本节先分析现有现有图神经网络在无属性图上的局限性，进而分析如何解决分析所得局限性。最后，本节通过理论分析证明迭代筛选算法有效的解决了现有工作的局限性。

现有图神经网络在无属性图上的局限性　现有的无属性图节点表示方法要么为节点分配随机标签[104,105]，要么为所有节点分配相同的嵌入向量[52,103]。当它们应用于无属性图的节点分类问题时，它们至少会受到以下两种局限性之一的影响：1）节点区分性和 2）

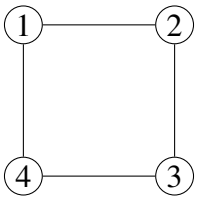

图 5.5　四个节点的正方形图 C_4

等变性属性。

节点区分性

本节先考虑节点区分性。现有的方法[52,53,103] 为所有节点分配相同的嵌入向量，因此导致现有的模型无法有效区分不同的节点信息。考虑一个常见的图卷积神经网络（Graph Convolutional Network，GCN），它通过一组全连接层对每个节点向量与其相邻节点向量进行编码来学习该节点的隐含向量表示。在图 5.5 给出的示例当中，使用此方法会造成所有的四个节点都将具有相同的隐含向量表示（因为每个节点都由相同的嵌入向量表示，同时，所有节点都具有相同的相邻节点信息）。

其他为所有节点分配随机标签的方法[104,105]则引入了不合理的人工标签。因为这些标签所对应的嵌入向量不能代表不同的训练样本之间所共享的语义知识，同时，这类向量的编码也无法推广到新样本之上。

等变性属性

等变性属性被认为是各种图神经网络分类任务中的必备属性[106,107]。

在无属性图的节点分类任务中，如果节点的位置信息发生变化，输出也会相应变化，如公式 (5.3) 所示。因此，正如 Wu et al.（2021）[108] 所建议的那样，为节点分类任务设计一个满足公式 (5.5) 等变性的图神经网络是很诱人的。否则的话，如果图神经网络不满足某种形式的等变性，图神经网络将对不合理的人工标签进行学习，从而影响训练[104,105]。以前的工作通过对所有节点使用相同的嵌入来实现等变性属性 (5.5)[52,53,103]。

然而，本节在此表明，满足公式 (5.5) 等变性的图神经网络将在多合理输出的节点分类任务上失效。换句话说，从 X 到 Y 的映射如果不是一个函数，并且给定一个输入 X，存在多个 Y 使得 $H(X,Y)$ 成立。通常，图神经网络通过一个函数 $Y = f(X)$ 来预测一个合适的 Y。

本节用一个非平凡自同构图的例子来展示满足等变性的图神经网络的局限性，即存在一个非平凡变换 π 使得 $\pi(X) = X$。如果公式 (5.5) 成立，则

$$\pi(X) = Xf(X) = \pi(f(X)) \tag{5.8}$$

这意味着所有被 π 变换所影响的节点，其对应的图神经网络的输出必须相同。

不幸的是，这对于各种任务来说可能是一个不合适的解决方案。本章在这里考虑

最大独立集 (MIS) 问题，它的目的是选择最大数量的不互相连接的顶点作为一个集合。以图 5.5为例，$\{1,3\}$ 是一个最大独立集，$\{2,4\}$ 也是一个最大独立集。然而，在这个例子中，满足等变性的图神经网络无法成功预测任意一个最大独立集。其原因是因为存在一个变换 π（例如，$\pi: 1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 4, 4 \mapsto 1$）使得所有节点的输出都必须相同。

本节通过一个非平凡自同构的例子来展示现有工作的局限性。然而，本节的分析对实际应用同样具有启发性，因为无属性图通常具有非常相近的局部图结构。

迭代筛选算法对现有工作的局限性的缓解 本节进一步分析迭代筛选算法与上述局限性的关系。迭代筛选算法有效的缓解了现有工作的局限性。

节点区分性

针对节点区分性，迭代筛选算法通过随机对节点进行标记来增加节点的可区分性。然而，直接对节点进行随机标记会使得节点具有不合理的人工标签，从而影响训练。因而，迭代筛选算法进一步使用多种不同节点标记的集合平滑来不合理的人工标签问题。对于训练来说，其图上各节点标签是随机分配得到的。这可以作为一种数据增强的方式，即通过集成学习的方式对数据做出平滑。通过这种方式，迭代筛选算法缓解了现有工作中节点区分性这个局限性。

等变性属性

本节先分析不受该局限性影响的图神经网络应该满足的性质。基于分析所得性质，本节进一步证明迭代筛选算法满足该性质，即不受现有等变性属性局限性的影响。

满足公式 (5.5) 等变性的图神经网络对无属性的节点分类存在局限性。其原因是它不合理地假设输出是输入的函数，即满足 (5.4)。

因此，本节放宽了对多合理输出节点分类任务中这一等变性约束，并在此设置下分析了所需的等变性。本节将 $\mathcal{H}(X) = \{Y : H(X,Y)\}$ 表示为给定图 X 的所有正确输出的集合。训练集通常为特定图 X_* 提供一个参考的解决方案 $Y_* \in \mathcal{H}(X_*)$，因为在实际应用中通常一个参考的解决方案已经足够（这和图神经网络的输入输出性质同时也是函数的输入输出性质恰好相同）。

将 $\mathcal{H}_*(\cdot)|_{\mathcal{D}}$ 定义为 $\mathcal{H}(\cdot)$ 的最小等变子集，使得 $Y_* \in \mathcal{H}_*(X_*)$，其相应的域限制为

$$\mathcal{D} = \{X : X = \pi(X_*) \text{ for some } \pi \in S_n\} \tag{5.9}$$

从定义

$$\mathcal{H}_*(X_*) = \{\gamma(Y_*) : \gamma \in S_n \text{ and } \gamma(X_*) = X_*\}, \tag{5.10}$$

出发，由于自同构变换 γ 的存在，它从本质上支持除给定 Y_* 之外的多个正确输出。

如果对于 π 满足，$X = \pi(X_*)$ ，等变性性质表明

$$\mathcal{H}_*(X) = \pi\left(\mathcal{H}_*(X_*)\right) \tag{5.11}$$

在这里，定义

$$\pi(\mathcal{H}_*(X_*)) \triangleq \{\pi(Y) : Y \in \mathcal{H}_*(X_*)\} \tag{5.12}$$

本节期望设计一个正确预测合理输出的图神经网络，即

$$f(X_*) \in \mathcal{H}_*(X_*). \tag{5.13}$$

由于等变性，可以得到

$$f(\pi(X_*)) \in \mathcal{H}_*(\pi(X_*)) = \pi(\mathcal{H}_*(X_*)). \tag{5.14}$$

根据 (5.10) 中 $\mathcal{H}_*$ 的定义，公式 (5.13) 意味着存在 $\gamma_1 \in S_n$ 使得

$$\gamma_1(X_*) = X_* f(X_*) = \gamma_1(Y_*) \tag{5.15}$$

同样，公式 (5.14) 意味着存在 $\gamma_2 \in S_n$ 使得

$$\gamma_2(X_*) = X_* f(\pi(X_*)) = \pi\gamma_2(Y_*) \tag{5.16}$$

结合这些公式，并用 γ 来表示 $\gamma_2\gamma_1^{-1}$，则存在 $\gamma \in S_n$ 使得

$$\gamma(X_*) = X_* \quad \text{和} \quad f(\pi(X_*)) = \pi\gamma(f(X_*)). \tag{5.17}$$

本节称 (5.17) 为 广义等变性。实际上，公式(5.5) 是公式(5.17) 在 γ 作为一个恒等变换时的特例。然而，本章通过在解空间中允许额外的自同构变换 γ 来满足公式 (5.5)，因此它不受第 5.1.1.3节中所分析问题的限制。

该分析表明,用于无属性节点分类图神经网络技术应满足公式(5.17) 而不是公式(5.4)。

对于迭代筛选算法来说，如果有足够数量的采样变换，其的预测过程会渐近地满足节点分类的广义等变性的条件。同时，本节还分析了迭代筛选算法与 EM 算法之间的联系。

在推断过程中，迭代筛选算法为一个图分配多个标签，并选择具有最高预测概率的预测作为输出。在形式上，带有标签 τ 的图 X 的联合预测概率可以表示为

$$s(X, \tau) = \prod_{i=1}^{n} \max_{j=1,\cdots,k} f_{ij}(\tau(X)) \tag{5.18}$$

其中 f 表示一个图神经网络函数，其输出一个 $n \times k$ 大小的矩阵（n：图节点数，k：分

类类别数)。其中，第 i 行是第 i 节点的预测分布。

在预测的过程中，迭代筛选算法对于多个标签 τ，选择使得概率 $s(X,\tau)$ 最大的标签分配作为最佳标签，其计算方式由下式给出

$$\tau_*^{(X)} = \underset{\tau \in S_n}{\operatorname{argmax}}\, s(X,\tau) \tag{5.19}$$

迭代筛选算法的预测过程公式是

$$\hat{Y}(X) = \left(\tau_*^{(X)}\right)^{-1}\left(f\left(\tau_*^{(X)}(X)\right)\right) \tag{5.20}$$

该公式遵循本章所定义的邻接矩阵表示规定。当迭代筛选算法分配节点标记时，本章通过 $\tau_*^{(X)}$ 变换 X 和 Y 的索引。在图神经网络处理之后，本章需要一个逆变换 $\left(\tau_*^{(X)}\right)^{-1}$ 来获得对 X 的预测，因为迭代筛选算法的预测应该对应于原始输入的图 X，而不是变换后的 $\tau_*^{(X)}(X)$。

定理 1：$\hat{Y}(\cdot)$ 满足广义等变性, 即, 对于任意图 X 和其对应的变换 $\pi \in S_n$, 存在一个 $\gamma \in S_n$ 使得

$$\gamma(X) = X \text{ 且 } \hat{Y}(\pi(X)) = \pi\gamma(\hat{Y}(X)) \tag{5.21}$$

证明：考虑任意无属性图 X 和其变换 π. 将公式 (5.19) 中的 X 替换为 $\pi(X)$，得到

$$\tau_*^{(\pi(X))} = \underset{\tau \in S_n}{\operatorname{argmax}}\, s(\pi(X),\tau). \tag{5.22}$$

此时，公式 (5.22) 和公式 (5.19) 本质上是同一个问题，并且它们的最优值应该由相同的计算得到，即

$$\tau_*^{(X)}(X) = \tau_*^{(\pi(X))}\pi(X) \tag{5.23}$$

这实质上意味着 $\tau_*^{(X)}$ 和 $\tau_*^{(\pi(X))}\pi$ 两个排列在 X 上会产生相同的结果，也就是说它们除开自同构变换外之间是相同的。换句话说，存在 γ 使得

$$\gamma(X) = X\tau_*^{(X)} = \tau_*^{(\pi(X))}\pi\gamma \tag{5.24}$$

其可以重新表示为

$$\tau_*^{(\pi(X))} = \tau_*^{(X)}\gamma^{-1}\pi^{-1}. \tag{5.25}$$

通过将公式 (5.20)中的 X 替换 $\pi(X)$，可以得到

$$\hat{Y}(\pi(X)) = (\tau_*^{(\pi(X))})^{-1}(f(\tau_*^{(\pi(X))}\pi(X))) \tag{5.26}$$

$$= \pi\gamma(\tau_*^{(X)})^{-1}(f(\tau_*^{(X)}\gamma^{-1}\pi^{-1}\pi(X))) \tag{5.27}$$

$$= \pi\gamma(\tau_*^{(X)})^{-1}(f(\tau_*^{(X)}(X))) \tag{5.28}$$

$$= \pi\gamma(\hat{Y}(X)), \tag{5.29}$$

其中 (5.27) 是引入了公式 (5.25)中的替换得到；(5.28) 是由 π^{-1} 和 π 相乘和 γ 的自同构性质（即 $\gamma(X) = X$）计算得到；而(5.29)是由 $\hat{Y}$ 在公式 (5.20)中的定义得到。

迭代筛选算法同时也与 EM（Expecatation–Maximization）算法相关。

定理 2：迭代筛选算法的训练过程是一种具有先验 S_n 的硬 EM 算法。

证明：可以将标记 π 视为将图 X 映射到输出 Y 的任务中的隐变量。公式 (5.6) 中的 min 运算符是求满足在 $\pi \in S_n$ 时给定输入和隐变量的其输出所得最大的概率值，将此概率值用 $P(Y|X,\pi)$ 表示。假设对于 $\pi \in S_n$ 有先验概率 $P(\pi)$，这相当于使用一个隐变量 π 进行交叉熵训练，该隐变量目标是最大化后验概率 $P(\pi|X,Y) \propto P(Y|X,\pi)P(\pi)$，即硬 EM 算法[109]。

这个简单的定理为迭代筛选算法方法提供了进一步的分析。EM 算法以处理分布的多模态混合而闻名，其类似于多合理输出的节点分类问题。迭代筛选算法的训练也类似于 EM 算法中的 E 步，它决定了样本对混合成分的适应度。因为 S_n 的搜索空间很大，迭代筛选算法采用硬 EM 算法的变体，它选择单个最佳变换进行训练。此外，迭代筛选算法假设 S_n 具有一个统一的先验标签，以这种方式可以消除不合理的人工标签。

5.1.2　实验验证

本章在最大独立集求解任务和布尔表达式求解任务上进行了实验，选择了最先进图神经网络架构作为基础模型，并将迭代筛选算法与各种节点知识编码策略进行了比较。

5.1.2.1　对比方法

静态标签分配　静态标签分配根据节点的身份信息进行嵌入表示（例如，图 5.1中的 x_1 和 c_1），尽管这样的身份信息并不代表有实际意义的语义信息。静态标签分配的方法在之前的工作中被广泛应用[104,110]。

同嵌入　该方法为无属性图中的所有节点分配相同的嵌入。这是在之前的工作中采用的[52,53]方法，用以消除节点上不合理的人工标签。

随机标签分配 随机标签即第三章中的随机生成算法，该方法在训练和预测的过程中随机分配嵌入。该方法也是迭代筛选算法的一个特例，即当 $K = 1$ 时，迭代筛选算法会退化为该方法。

度特征 在不使用不合理的人工标签的情况下对节点进行编码的一种方法是通过其度数信息来捕获节点的一些局部信息。在此方法中，节点表示使用 $1/(d_v + 1)$ 作为一维、不可学习的嵌入特征，其中 d_v 是节点 v 的度数。

度排序嵌入 使用度特征信息作为单个数值特征的缺点是低维性和不可学习性。因而，度排序嵌入通过嵌入节点的度排名来扩展以上想法。具体来说，度排序嵌入按度数降序对所有节点进行排序，其中，度数排第 i 个的节点由第 i 个嵌入向量 $\boldsymbol{e}_i$ 编码。

5.1.2.2 实验 1：最大独立集求解问题

本章先在最大独立集 (MIS) 求解任务上验证迭代筛选算法方法。在图论中，一个独立集是一组节点集合，其中所有的节点两两之间并不相连。独立集是 最大独立集，当且仅当它在所有独立集中拥有数量最多的节点。最大独立集求解是一个 NP-hard 问题，旨在从图中找出最大独立集。

对于输入图，此任务中图神经网络模型的目标是预测每个节点的二分类标签，从而确定该节点是否在最大独立集当中。为了从模型预测中得出一个最大独立集，本章使用了一种简单的搜索算法。本章先根据节点在最大独立集中的预测概率对所有节点进行降序排序。然后，依次遍历每个节点，选择排在最前面的节点放入最大独立集；同时，将它的邻居从排序列表中删除。迭代该过程，直到处理完整个节点排序列表。这样就保证了被选中的节点是一个独立集。本章的实验验证进一步验证其是否为最大独立集。

模型 本实验采用最先进的模型（GCN[52]）实现最大独立集求解。Li et al.（2018）[52] 对所有节点使用相同的嵌入。该模型包含 20 个图卷积层，通过 dropout 技术以 0.1 概率进行正则化。对于该模型中使用的所有层的隐含层大小，本实验将其设置为 128 维。对于训练，本实验使用 Adam[80]优化器在单个 Titan RTX 上以 10^{-4} 的学习率进行模型训练。

数据集 本章遵循之前工作中的数据合成过程[52]，分别生成了 173,751、20,000、20,000 个图用于训练、验证和测试。图中的节点数由均匀分布 $U[100, 150]$ 生成。

表 5.1　MIS 求解问题的实验结果（‘‘迭代筛选算法 -10” 表示 $K = 10$，随机采样 10 个不同的变换进行训练和预测）

行 #	GCN[52]	准确率
1	同嵌入	75.59%
2	度特征	73.22%
3	度排序嵌入	71.58%
4	静态标签分配	74.57%
5	随机标签分配	75.28%
6	迭代筛选算法 -10	**85.04**%

表 5.2　SAT 求解问题的实验结果（‘‘测试-k” 表示使用 k 个变量的测试集. ‘‘预测-m” 表示在预测时随机采样了 m 个不同变换）

		错误率				
行 #	NLocalSAT[53]	测试-5	测试-10	测试-20	测试-40	平均值
1	同嵌入	5.26%	8.17%	15.03%	27.62%	14.02%
2	度特征	5.31%	8.37%	14.25%	24.94%	13.22%
3	度排序嵌入	5.45%	10.23%	16.17%	28.04%	14.97%
4	静态标签分配	6.11%	9.86%	16.89%	28.88%	15.44%
5	静态标签分配 & 预测-10 (平均值)	5.00%	8.77%	15.74%	29.70%	14.80%
6	静态标签分配 & 预测-10 (最大概率)	1.77%	3.65%	7.86%	16.22%	7.38%
7	随机标签分配	3.38%	6.17%	12.70%	23.66%	11.48%
8	随机标签分配 & 预测-10 (平均值)	3.39%	6.07%	12.42%	23.34%	11.31%
9	随机标签分配 & 预测-10 (最大概率)	2.72%	5.03%	11.37%	22.06%	10.30%
10	迭代筛选算法 -10 (最大概率)	**1.13**%	**1.68**%	**1.81**%	**5.24**%	**2.47**%

评价指标　本章的评价指标使用了求解准确率。由于上述的后处理确保输出一定是一个独立集，因此只需评估它是否和参考解具有相同数量的节点。如果预测的独立集与参考最大独立集具有相同数量的节点，则说模型正确地解决了这个输入数据对应的最大独立集问题，否则，则认为模型预测出错。

实验结果　表 5.1 展示了最大独立集求解问题的实验结果。

可以观察到静态标签分配（第 4 行）的性能较低（74.57%），因为它引入了不合理的人工标签。相同和随机标签分配（第 1 和 5 行）消除了此类不合理的人工标签并具有更好的效果（75.28%）。度特征（第 2 行）和度排序嵌入（第 3 行）受到第 5.1.1.3节中提到的等变性属性的限制，并且表现比其他方法效果差。

相比之下，迭代筛选算法（第 6 行）能够消除不合理的人工标签，同时实现在公式 (5.17) 中所提出的广义等变性性质。它的性能高于所有对比方法，其预测问题数量比效果最好的对比方法（同嵌入方法，75.59% 的准确率）还要下降 39%。

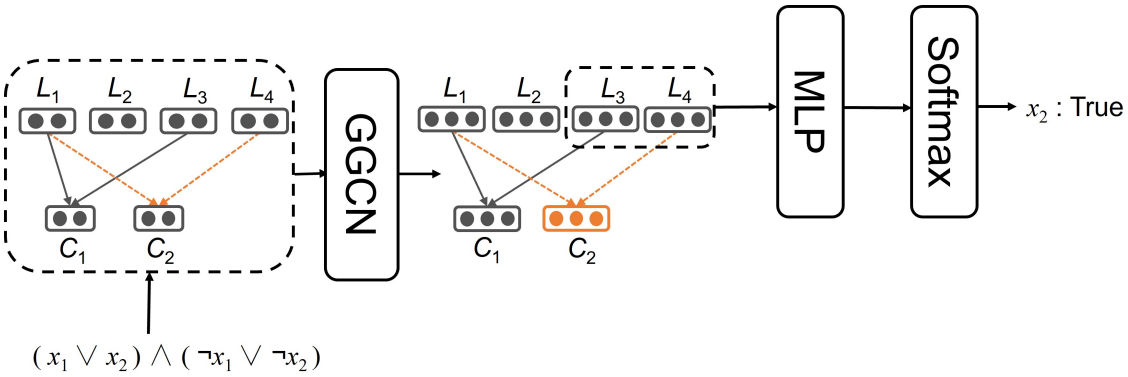

图 5.6　NLocalSAT 算法[53]

5.1.2.3　实验 2: SAT 求解问题

本章在 SAT 求解问题上进一步评估迭代筛选算法。布尔可满足性问题（SAT）是计算机科学中最基本的问题之一。给定一个命题公式，如果存在一个命题变量的真或否赋值分配能使得整个公式的结果为真，这样的赋值分配被称为一个证书，这样的一个命题公式被称为可满足的。

本实验考虑 SAT 求解问题的一个特定设置。假定已知给定公式是可满足的，而本实验的目标是通过图神经网络来预测其对应证书，即应该为每个变量赋值为真或否。这是 SAT 求解器的关键步骤。

模型　本实验对于图神经网络模型和设置，采用最先进的 NLocalSAT[53]模型及其设置。在该模型中，SAT 公式表示为二分图，其中节点是子句或是文字（参见图 5.1 示例）。节点由标识符表示，标识符是人工标签。

NLocalSAT 模型[53]如图 5.6所示，其使用了门控图卷积的方法（GGCN）对输入的 SAT 图进行处理并预测每个变量的具体取值。Zhang et al.（2020）[53] 对所有子句/文字节点使用相同的嵌入表示。该模型有 16 个相同的门控图卷积层，通过 0.1 概率的 dropout 进行正则化。本实验分别对 NLocalSAT 中文字和子句使用不同离散空间的嵌入向量来表示。

数据集　本章使用了论文 Zhang et al.（2020）[53] 中所给出的生成器来生成 SAT 数据集。该数据集的训练集和验证集分别包含 50 万和 39.6 万个 SAT 公式。该公式中的变量数量由一个均匀分布 $U[10,40]$ 生成，而其对应的子句数则由 $U[2,6]$ 生成。为了进行测试，测试集包含四组不同的难度级别。具体来说，每个测试集中一个输入公式的变量个数是 5、10、20 或 40 个，分别用测试-5、测试-10、测试-20 或测试-40 来表示，其中每个包含 4 万、2 万、1 万、或 5 千个 SAT 公式。

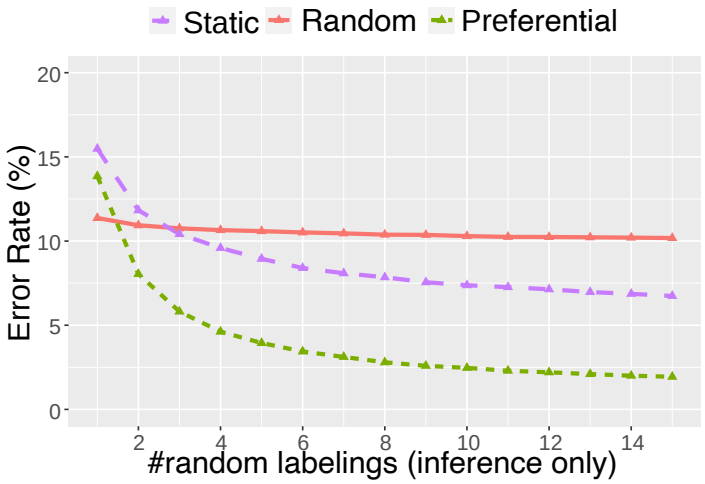

图 5.7　预测错误率（Y 轴）和预测期间所用的随机标签采样数（X 轴）之间的关系（本章比较了不同的方法的训练过程，同时所有的方法都是选择最大预测概率结果来作为最后的预测结果）

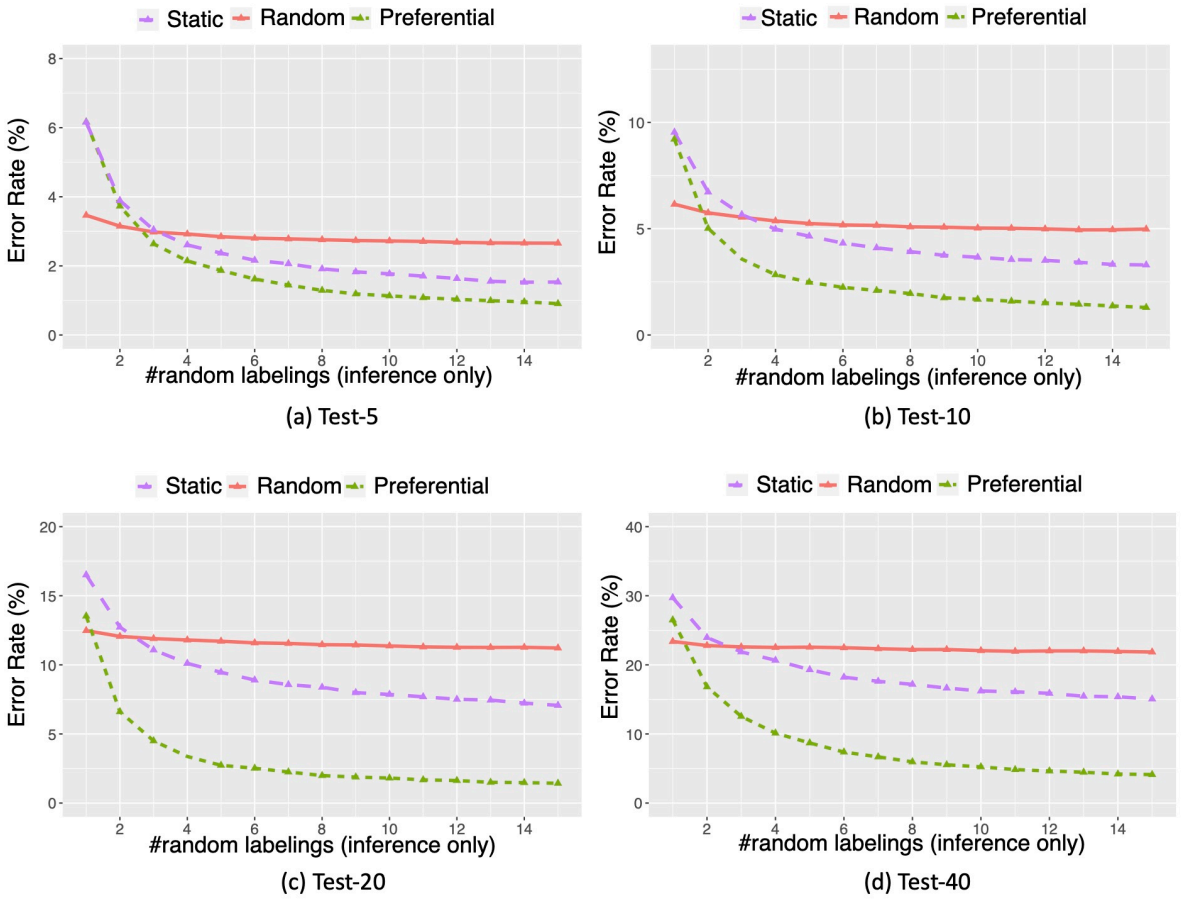

图 5.8　预测错误率（Y 轴）和预测期间所用的随机标签采样数（X 轴）之间的关系（本章在这里列出了不同测试集上的效果）

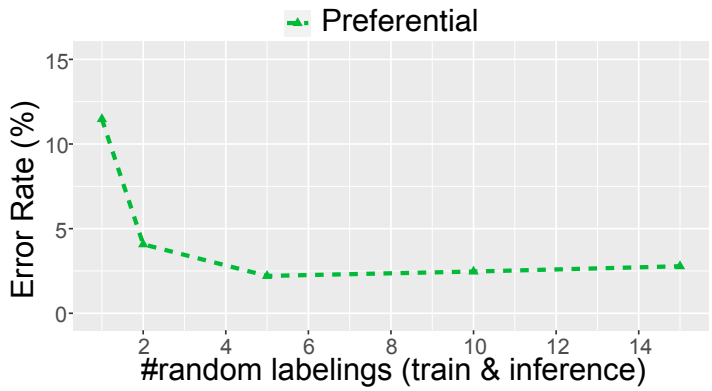

图 5.9　预测错误率（Y 轴）和训练及预测期间所用的随机标签采样数（X 轴）之间的关系

评价指标　模型的性能通过公式级别的错误率来评估，如果预测所得的赋值分配不能使输入的公式为真，则认为该模型预测错误。

实验结果　表 5.2 展示了 SAT 求解的实验结果，

如上文第 5.1.1.3节所述，静态标签（第 4 行）引入了节点上不合理的人工标签，而使用相同的嵌入表示（第 1 行）无法很好地区分不同的节点。度特征方法缓解了这些问题，并且在此任务中的表现优于第 1 行和第 4 行。但是，这些方法在总体上仍然表现不佳。

本实验进一步分析满足公式 (5.4) 等变性的图神经网络的性能。满足等变性既可以通过训练的随机标签（第 8 行和第 9 行）来实现，也可以通过在预测（第 5 行和第 8 行）期间显式引入平均集成的方式（即于对多个输出结果取平均值），或使用相同的嵌入（第 1 行）或基于度嵌入（第 2 行和第 3 行）。它们的性能虽然优于静态标签（第 4 行），但似乎仍旧不足。

然后，本实验评估迭代筛选算法在预测阶段的效果。相比于不同的对比方法，本章的方法放宽了公式 (5.5) 的要求，并在预测过程中满足广义等变性公式(5.17)。可以看到其错误率（第 6 行和第 9 行）远低于满足等变性公式的图神经网络。

此外，迭代筛选算法显式减少了训练期间的不合理的人工标签。在预测算法不变的情况下，迭代筛选算法在很大程度上优于使用静态标签和随机标签的训练方法（第 10 行对比于 第 6 和 9 行）。

本实验进一步分析了随机标签的采样数量如何影响推理过程中的模型效果，如图 5.7和图 5.8所示。可以观察到所有模型都通过更多的采样次数实现了更好的效果。然而，其对于随机标签分配的方法来说，改进是微不足道的，因为无论采样的数量如何，它从根本上受到等变性函数的限制。

如果采样数量很少（例如，≤ 2），静态标签分配和迭代筛选算法不会获得良好的

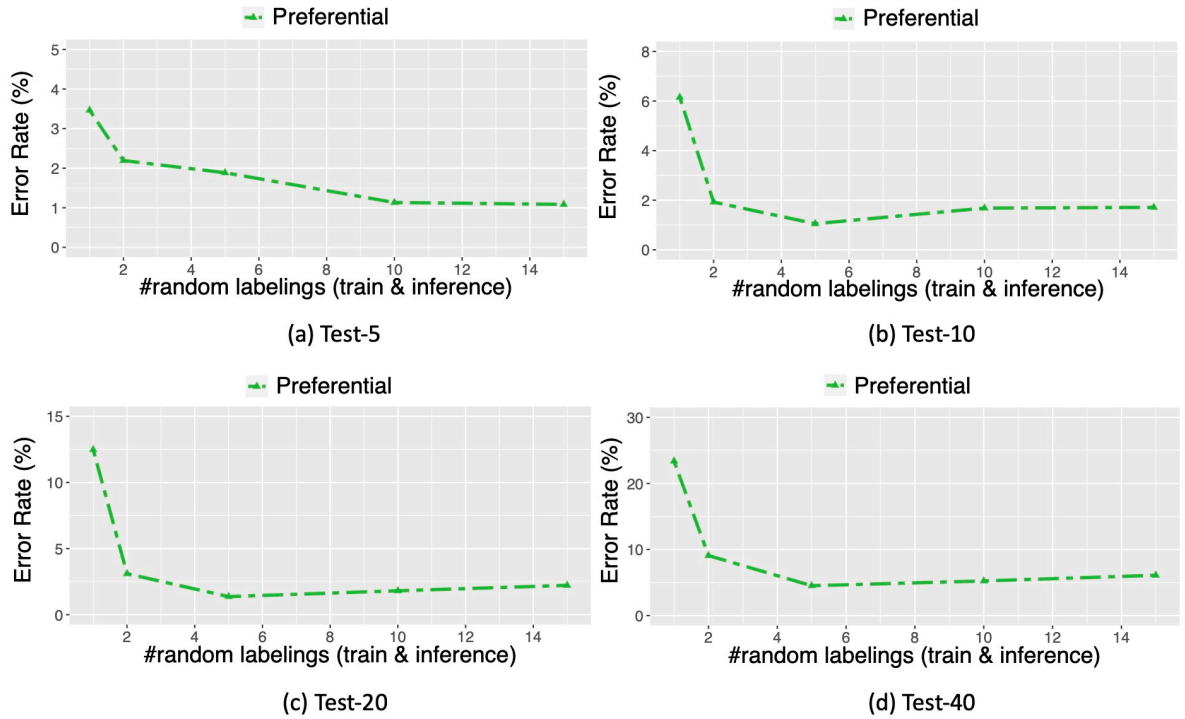

图 5.10　预测错误率（Y 轴）和训练及预测期间所用的随机标签采样数（X 轴）之间的关系（本章在这里列出了不同测试集上的效果）

效果。一个合理的解释是，在预测过程中的少量采样所得标签可能和训练时所用的标签分配不符。然而，当有更多的采样时，其效果会显著提高，因为这些模型能够放宽函数等变性但渐近地实现广义等变性。具体来说，迭代筛选算法始终比静态标签分配效果好很多，因为本章的模型是用硬 EM 算法方式以最佳标签数据训练的。

最后，图 5.9和图 5.10 中显示了训练和标签采样数量如何影响迭代筛选算法的性能。图 5.10使用不同数据集的错误率作为衡量指标，而图 5.9使用错误率的平均值作为衡量指标。结果表明，如果采样的数量为 1，迭代筛选算法则会退化为随机标签分配算法，从而得到较差的性能。然而，当数量增加时，其预测错误率显著下降，当数量大于或等于 5 时，错误率变得稳定。该实验表明即使当计算资源受到限制时，迭代筛选算法仍然可以应用。

5.2　基于检测和修复算法的机器翻译提升任务

随着机器学习的发展，深度学习技术已经为用户提供了通用的自然语言翻译系统，许多翻译系统能够实时有效地在数千对语言之间进行翻译[111]。然而，这样的翻译系统并不完美，并且其问题与传统的、非基于机器学习的翻译系统的问题不大相同[112-115]。

例如，机器翻译常常出现出有害公平性的问题，这些问题对特定的用户群体造成

了严重的伤害[116]。本章在广泛使用的工业级机器翻译系统中发现了此类公平问题的部分示例。图 5.11 显示了几个这样的谷歌翻译结果（英语 → 中文）①。从图中可以看出，当主语是“男性”或“男学生”时，谷歌翻译会将“good”翻译成“很好的”。然而，有趣但也令人遗憾的是，当主语是“女性”或“女学生”时，它会将“good”翻译成“很多”②。

这种翻译不一致可能会使用户感到困惑，而且对计算机科学领域的女性研究人员来说显然是“不公平”的；与进行“很好的”研究相比，进行“很多”研究显然是一种更贬义的解释。为了避免这种大规模的不公平的翻译，本章需要能够自动识别和纠正这种不一致的技术。

English	Chinese (Google Translation)	Notes
Men do good research in computer science.	Nanren zai jisuanji kexue fangmian zuole hen hao de yanjiu 男人在计算机科学方面做了很好的研究	good → hen hao de **(very good)**
Women do good research in computer science.	Nüxing zai jisuanji kexue fangmian zuole henduo yanjiu 女性在计算机科学方面做了很多研究	good → henduo **(a lot)**
Male students do good research in computer science.	Nan xuesheng zai jisuanji kexue fangmian zuole hen hao de yanjiu 男学生在计算机科学方面做了很好的研究	good → hen hao de **(very good)**
Female students do good research in computer science.	Nü xuesheng zai jisuanji kexue fangmian zuole henduo yanjiu 女学生在计算机科学方面做了很多研究	good → henduo **(a lot)**

图 5.11　从谷歌翻译器中所找出的影响公平性的翻译例子

上述不一致问题实质上是一种扰动约束感知问题，因而，本章将第四章所提出的检测和修复算法（CAT）引入到机器翻译当中。它可以自动为机器翻译系统检测不一致性问题，并自动修复所检测出来的不一致性问题进而提升翻译的可接受性。

5.2.1　检测和修复算法的拓展

为了适配机器翻译系统，检测和修复算法（CAT）在修复时所使用的词对齐技术为 Liu 和 Sun（2015）[117] 所提出的技术，该技术使用隐变量对数线性模型进行无监督的

① 这四个翻译是在 2019 年 7 月 23 日获得的。这些示例纯粹是为了说明目的，而不是批评谷歌翻译的质量。其他主流翻译技术很可能也会有类似的问题。
② 类似的问题也存在于其他语言之间的翻译当中。通过粗略的检查，本章已经找到了德语 → 中文的例子。

词对齐。同时，映射约束改为以下两种规则：

- 1）约束输入中的词语 w_i, w_i^r 和其对应翻译中的词语 $t(w_i), t(w_i^r)$ 必须属于同一类型（即数字或非数字）。
- 2）如果替换的词是非数字类型，本章应用 Stanford 解析器检查替换是否会导致结构变化。

其余结构与用于第 四章中检测修复算法（CAT）相同。

5.2.2 实验验证

本节将介绍验证 CAT 在自动检测输入生成、约束不一致性检测和约束不一致性修复上的实验设置。

5.2.2.1 实验设置

研究问题　本章的研究问题有 3 个：

- **RQ1：CAT 方法生成检测输入的准确率如何？**
  本章通过随机抽样一些生成的检测输入对，并进一步（人工）检查它们是否有效来回答这个研究问题。这个问题的答案目的是为了保证 CAT 确实生成了适合一致性检测及修复的输入。
- **RQ2：CAT 揭示不一致性问题的能力如何？**
  为了回答 RQ2，本章根据相似度指标计算一致性分数以作为不一致性检测的结果。为了评估 CAT 对不一致性问题的揭示能力，本章手动检查翻译样本，并将手动检查结果与自动测试结果进行对比。
- **RQ3：CAT 自动修复不一致性问题的能力如何？**
  为了回答这个问题，本章评估了修复不一致性问题的数量及比例。同时，本章还人工检查了由 CAT 修复的翻译，并检查它们是否提高了翻译的一致性和可接受性。

机器翻译系统　本实验同时考虑了工业级机器翻译系统和最先进的面向研究的机器翻译系统。一个是谷歌翻译[118]（在结果部分缩写为 GT），由谷歌开发的一种广泛使用的机器翻译系统。另一个是 Transformer[62]，学术界研究的通用机器翻译系统。

本章使用谷歌翻译，因为它是一个强制执行黑盒修复的系统示例；用户无法访问训练数据或翻译系统的代码，因此，根据定义，任何改进都只能通过黑盒方法来实现。当然，它也是一个高质量的主流工业级翻译系统，其使结果更有趣且更有说服力。

本章使用默认设置来训练 Transformer。Transformer 基于三个数据集进行训练：CWMT 数据集[119]（包含 7,086,820 个平行语料），联合国数据集[120]（包含 15,886,041 个平行语

料），和新闻评论数据集[121]（包含有 252,777 个平行语料）作为训练数据。验证数据（帮助调整超参数）也来自新闻评论数据集，包含 2,002 个平行语料。Transformer 的实现使用了 Tensor2Tensor 深度学习库[122]。为了获得更有效的翻译，本章为模型训练了 500,000 代。

测试集 和之前机器翻译相关研究一样[123,124]，本章使用来自新闻评论数据集的测试集[121]。该数据集用以检测谷歌翻译和 Transformer 模型。测试集包含不同于训练集和验证集中数据的 2,001 个平行语料。

实验参数设置 本章在测试过程中为每个输入句子生成最多 5 个有效变异体，在修复过程中为每个输入句子中生成最多 16 个有效变异体。本章在第 5.2.2.3节中进行了一个额外的实验来研究变异体数上限的影响。

本章的实验是在 256GB RAM 和四个总共包含 32 个内核的 Intel E5-2620 v4 CPU（2.10 GHz）的 Ubuntu 16.04 上进行的。本章使用的神经网络都是在八块单个 Nvidia Titan RTX（24 GB 内存）上训练和预测的。

5.2.2.2 实验结果

本节用实验结果来回答研究问题。

对输入生成的效果 (RQ1) 要回答这个问题，对于数据集中的每个句子（总共 2,001 个），本章分别使用 CAT-N 和 CAT-L 的方法生成变异体。每个变异体与原始句子配对以形成检测输入。然后本章记录数量（即检测输入的总数）、分布（即每个句子生成的检测输入的数量）、有效性（即检测输入中的变异和原始句子是否适合于用作不一致性问题检测）和多样性（即生成的变异体的类型）。

数量：

对于 2,001 个输入句子，CAT-N 通过同位替换生成了 11,045 个候选词，通过候选过滤（使用自动语义验证）进一步丢弃了 1,103 个。总共生成了 9,942 个变异体，这些变异体可以与原始句子配对，作为机器翻译的检测输入。对于 CAT-L，它通过单词替换生成 21,960 个变异候选，其中 17,268 个被候选过滤（使用 Standford 解析器）丢弃。它最终产生 4,692 个检测输入。

这些实验结果表明，对于轻量级的 CAT-L，由于忽略了原始输入句子的上下文信息，只有 21% 的变异候选通过了过滤。CAT-N 生成的检测输入明显多于 CAT-L。特别是，CAT-N 的同位替换增加了候选词通过语义验证的可能性。对于 CAT-N，90% 的变异候选通过过滤。

在下文中，本章将深入研究生成的变异体的分布、多样性和有效性。

分布：

本章使用小提琴图来展示为每个句子生成的检测输入的整个分布。图 5.12 显示了结果。在该图中，可以观察到大部分输入句子由 CAT-N 生成的检测输入数量达到 5（人工设置的上限）。但是，对于 CAT-L，许多句子的检测输入少于 5 个。

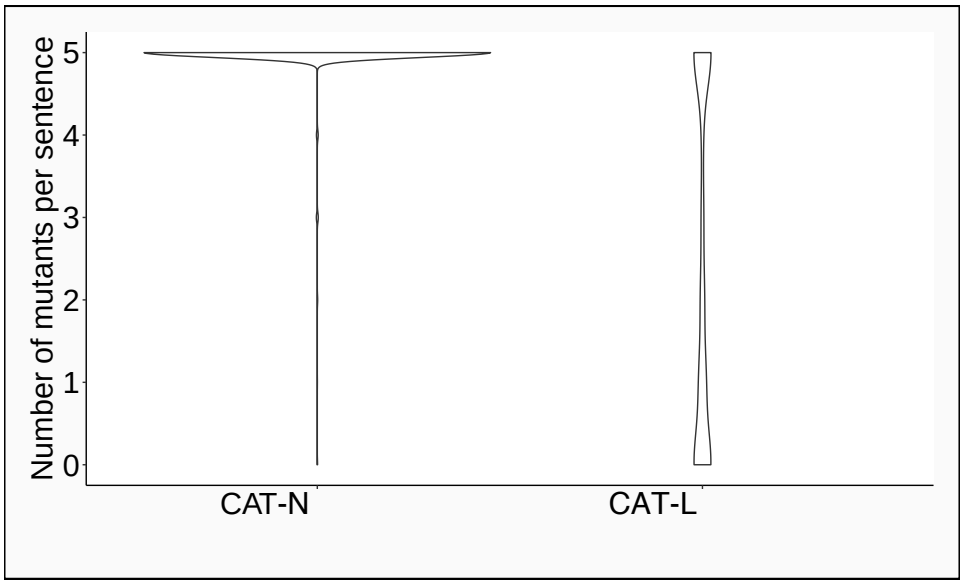

图 5.12　每个句子生成的变异体的分布（CAT-N 为几乎所有输入句子生成 5 个检测输入，而 CAT-L 未能为 43.7% 的句子生成任何检测输入）

特别是，对于轻量级的 CAT-L，它无法为多达 43.7% 的句子生成检测输入。因此，CAT-L 无法检测到这些句子的任何翻译问题。而使用同位替换的 CAT-N 能够为几乎所有句子生成检测输入（只有三个句子，即 0.15%，没有生成的检测输入）。

多样性：

本章从变异多样性方面进一步说明 CAT-N 和 CAT-L 的效果，即根据替换词的词性生成变异体的类型。词性类型包括名词、形容词、副词、数词、动词、限定词、连词、代词、介词等。

图 5.13 显示了属于每种类型的变异体数量。总的来说，可以观察到大多数 CAT-L 生成的变异体是用于名词、形容词和数词的，如第 4.1.2节中所述。这是因为 CAT-L 高度依赖预定义的变异算子，从而将变异类型限定至这三种。

相比之下，CAT-N 可以生成另外七种词性的变异体。这种展示的多样性很重要，因为不同类型的检测输入揭示了不同类型的不一致问题。例如，CAT-L 会错过对动词引起的不一致问题的检测，而动词对句子翻译的理解起着关键作用。

有效性：

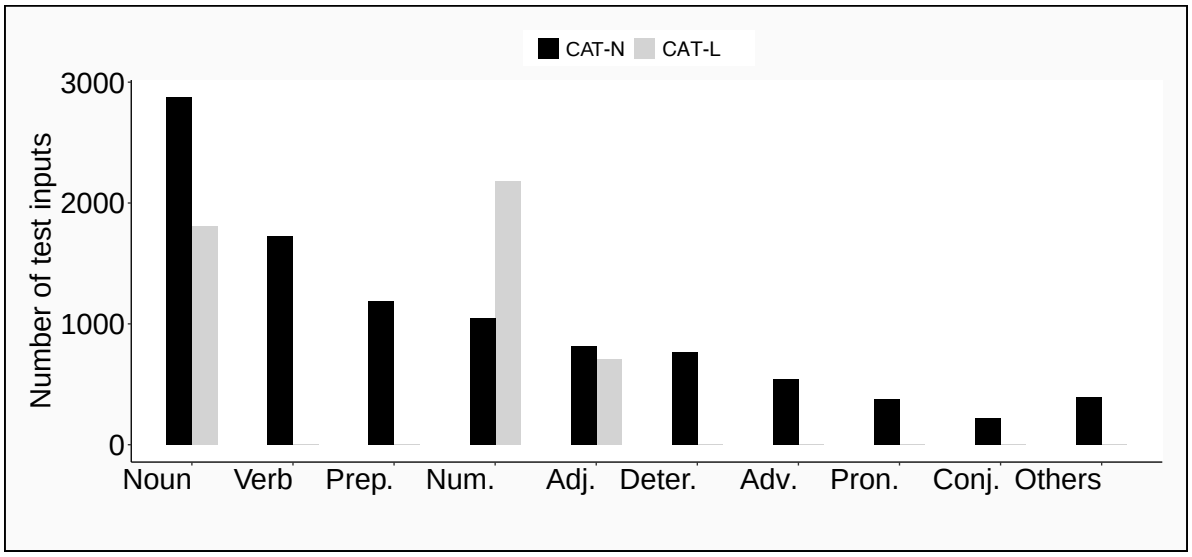

图 5.13　每个变异类型（x 轴）生成的检测输入（y 轴）的数量（该图显示 CAT-N 生成的变异体比 CAT-L 更多样化）

有效性衡量生成的检测输入句子符合翻译测试要求的比例。如果生成的变异体 1) 语法正确；2）语义上合理；3）产生与原始句子语义相同的翻译（不包含替换的单词的翻译），则认为检测输入是有效的。本章为 CAT-N 和 CAT-L 分别随机抽取 200 个测试用例。

人工检查① 表明 CAT-N 生成的检测输入中有 96.5% 是有效的。无效的变异示例是 1) "*Greater (protection→priority) should be given to whistleblowers, Sir Eric says;*" 2) "*The (Supplement→Suppression) of Public Sports Facilities;*"。对于 CAT-L，其检测输入的 93.0% 是有效的。其无效的检测输入为 1) "*He was a kind spirit with a big heart*: kind → sort;" 2) "*Two earthquakes with magnitude 4.4 and 4.5 respectively*"：Two → Six; 3) "*It is in itself a great shame*"：great → good. 这些结果表明 CAT-N 相比于轻量级的 CAT-L 不仅生成了更多的检测输入，而且还生成了更多合格的检测输入，用于检测不一致问题。

CAT-N 方法使用语义验证来过滤候选词以提高变异有效性。因此，本章也需要研究语义验证的有效性。事实证明，如果没有语义验证，10.0% 的检测输入无效②。通过语义验证，这个比例降低到 3.5%。因此，语义验证技术移除 (10% - 3.5%) / 10% = 65% 无效检测输入。

本章进一步研究了不同类型变异体的检测输入的有效性。表 5.3 显示了实验结果。在每个单元格中，第一个/第二个数字是属于相应类型的有效检测输入数/总输入数，括号中的比率是有效检测输入所占比例。有趣的是，可以观察到"形容词"变异类型对 CAT-N 和 CAT-L 的有效性最低。这一观察显示了在机器翻译检测中进一步提高检测输

① 卡帕分数平均为 0.96，这表明人工检查结果高度一致。
② 卡帕分数为 0.86.

入生成有效性的可能性。

表 5.3　不同类型变异体的检测不一致性问题的有效性

方法	名词	形容词	副词	数词	动词
CAT-L	83/92 (90%)	16/20 (80%)	0/0 (0.0%)	86/87 (99%)	0/0 (0.0%)
CAT-N	65/69 (94%)	11/14 (79%)	11/11 (100%)	18/18 (100%)	29/29 (100%)
方法	限定词	连词	代词	介词	其他
CAT-L	0/0 (0.0%)	0/0 (0.0%)	0/0 (0.0%)	1/1 (100.0%)	0/0 (0.0%)
CAT-N	18/18 (100%)	6/6 (100%)	5/5 (100%)	24/24 (100%)	6/6 (100%)

总的来说，对于 RQ1 ，有以下结论：

对 RQ1 的回答：CAT-L 可以有效生成 4,692 个检测输入，其中覆盖了 66.3% 的输入句子和 3 种类型的变异体，其生成输入的有效率为 93.0%。CAT-N 在生成的检测输入的数量、分布、多样性和有效性方面优于 CAT-L。特别是，CAT-N 生成 9,942 检测输入，覆盖 99.9% 的输入句子和 10 种类型的变异体，生成输入的有效率为 96.5%。

CAT 在揭示一致性问题方面的能力 (RQ2)　本节回答 RQ2，即验证 CAT 的不一致性问题的揭示能力。为了回答这个问题，本章先验证：1）变异体和原始翻译之间的一致性度量值；2）在不同角度下的一致性揭示结果。本章还探讨了一致性指标和人工检查在评估不一致性揭示方面的接近程度。

一致性度量值

本章使用谷歌翻译和 Transformer 翻译了前一个实验所生成的输入，并按照算法 4 的步骤将它们与原始句子的翻译进行比较（在这里，本章只使用了轻量级的 CAT-L 方法所生成的输入结果，其原因是 CAT-N 的结果与 CAT-L 相近）。对于每个变异体，本章计算四种一致性分数，每种分数对应于一个相似性度量（在第 4.2.1.2节中概述）。

图 5.14 显示了一致性分数低于 1.0 的直方图。从图中可以看出，不同的度量值具有不同的分布，但总体而言，所有四个相似性度量指标都报告了大量的翻译（即，大约 47% 的总翻译）低于 1.0 的分数，其表明存在翻译不一致问题。

表 5.4 显示了不同相似性度量阈值的报告不一致翻译的结果。从表 5.4 中可以看到即使对于非常宽松（低）的一致性阈值，其仍然存在不一致问题。

手动检查不一致性

此外，本章随机抽取了 300 个变异体的翻译。其中 2 个由于句子原因无法被读懂

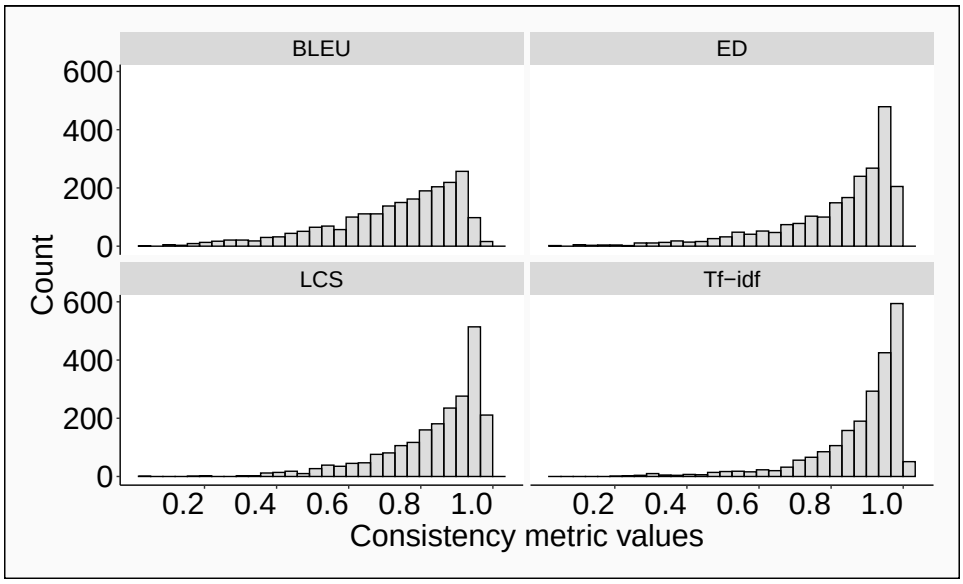

图 5.14　不一致性分数的分布直方图。大量翻译的相似度得分低于 1，表明存在许多不一致的翻译（RQ2）

表 5.4　在 1.0 和 0.6 之间具有不同阈值的报告的不一致问题的数量（使用 1.0 阈值，如果检测到任何不一致，则翻译被视为有不一致问题。阈值越低，认定不一致性问题的标准就越宽松。可以看到即使对于非常宽松（低）的一致性阈值，其仍然存在不一致问题）

	阈值	1.0	0.9	0.8	0.7	0.6
GT	LCS	2,053 (44%)	865 (18%)	342 (7%)	123 (3%)	57 (1%)
	ED	2,053 (44%)	913 (19%)	401 (9%)	198 (4%)	101 (2%)
	Tf-idf	2,459 (52%)	548 (12%)	208 (4%)	71 (2%)	21 (0%)
	BLEU	2,053 (44%)	1,621 (35%)	911 (19%)	510 (11%)	253 (5%)
Transformer	LCS	2,213 (47%)	1,210 (26%)	634 (14%)	344 (7%)	184 (4%)
	ED	2,213 (47%)	1,262 (27%)	700 (15%)	428 (9%)	267 (6%)
	Tf-idf	2,549 (54%)	851 (18%)	399 (9%)	188 (4%)	112 (2%)
	BLEU	2,213 (47%)	1,857 (40%)	1,258 (27%)	788 (17%)	483 (10%)

并分析，因此本章使用剩余的 298 个翻译进行分析。对于每个变异体，本章手动检查其翻译和原句的翻译。当满足以下任一条件时，将报告为不一致性问题：除了变异的替换词外，两个翻译 1) 具有不同的含义；2）有不同的感情色彩；3) 专有名词使用不同的翻译结果。

人工检查发现谷歌翻译有 107 (36%) 个翻译不一致，Transformer 有 140 (47%) 个翻译不一致①。

指标与人工检查之间的相关性

本章比较相似度度量分数和人类一致性评估结果。本章根据人工检查将 298 个人

① 谷歌翻译/Transformer 的卡帕值为 0.96/0.95，说明人工检查结果高度一致。

工标记的翻译分成两组。一个被标记为一致的翻译，另一个被标记为不一致的翻译。然后本章检查每组中的相似度度量值分数。

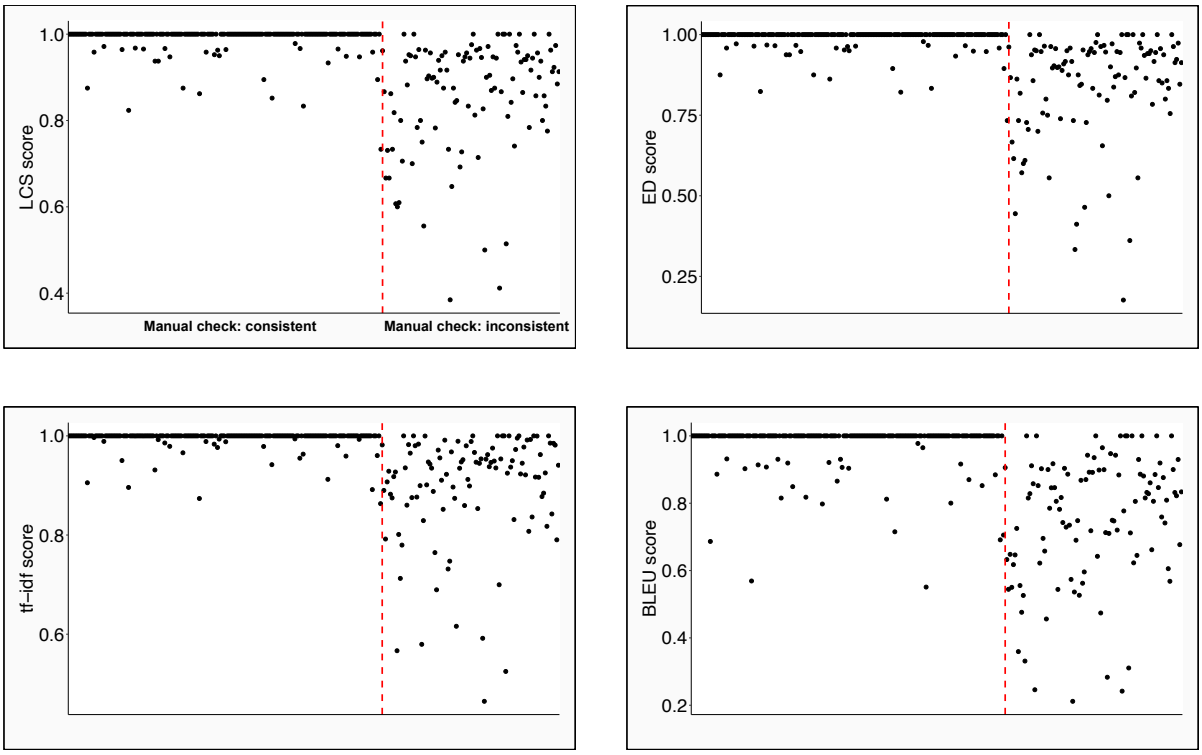

图 5.15　相似度度量分数的比较和翻译一致性的人工检查（相似度度量得分值和人工评估之间有很好的一致性，RQ2）

图 5.15 显示结果[①]。垂直虚线左侧/右侧的点描绘了手动标记为一致/不一致的翻译的相似度度量值。本章观察到左侧部分的大多数点 (82.2%) 得分为 1.0。左侧部分的得分值（平均值为 0.99）通常高于右侧部分（平均值为 0.86）。这些观察结果表明，度量值和人工检查倾向于就翻译不一致问题达成一致。值得注意的是，图 5.15 还显示了左侧部分存在一些低度量值的数据而右侧部分则存在一些高度量值的数据，这表明存在使用相似度度量来评估翻译一致性仍然存在误报和漏报的可能性。

相似度指标的阈值学习

相似度指标分数是连续值。如果需要自动报告不一致性问题，本章需要为每个指标设置一个单独阈值。本章目标是一个使相似度度量值判断最接近人工检查结果的阈值。为此，本章从谷歌翻译中随机抽取了另外 100 个翻译，并手动将它们标记为一致与否。然后，本章使用这 100 个标签通过遍历的方式来选择一个相似度阈值（从 0.8 到 1.0，步长为 0.01）以使得每个相似度度量的 F1 分数最大。以这种方式确定的四个相似度度量指标 LCS、ED、tf-idf 和 BLEU 的最佳阈值分别为 0.963、0.963、0.999、0.906，

① 本章仅分析 LCS 的结果

其对应的 F1 分数分别为 0.81、0.82、0.79、0.82。当度量值低于此阈值时，本章的方法将报告其翻译为一个不一致问题。

表 5.5　不一致性检测的精确率和召回率 (RQ2)

	指标	TN	FN	FP	TP	精确率	召回率	F1
GT	LCS	169 (57%)	16 (5%)	22 (7%)	91 (31%)	0.81	0.85	0.83
	ED	169 (57%)	16 (5%)	22 (7%)	91 (31%)	0.81	0.85	0.83
	tf-idf	162 (54%)	12 (4%)	29 (10%)	95 (32%)	0.77	0.89	0.82
	BLEU	171 (57%)	20 (7%)	20 (7%)	87 (29%)	0.81	0.81	0.81
Transformer	LCS	142 (48%)	22 (7%)	16 (5%)	118 (40%)	0.88	0.84	0.86
	ED	142 (48%)	21 (7%)	16 (5%)	119 (40%)	0.88	0.85	0.87
	tf-idf	141 (47%)	11 (4%)	17 (5%)	129 (43%)	0.88	0.92	0.90
	BLEU	147 (49%)	23 (8%)	11 (4%)	117 (39%)	0.91	0.84	0.87

为了了解所选择的阈值能否很好地捕捉到一致性和不一致之间的区别，本章分别在谷歌翻译和 Transformer 上使用 298 个先前采样的翻译来测试阈值。结果显示在表 5.5 中。假阳性（表中的 FP）表示阈值判断翻译不一致但人工检查是一致的。假阴性（表中的 FN）表示阈值判断翻译一致但人工检查不一致。从表中可以看出，假阳性和假阴性的比例都在 10% 以下，本章认为其效果是可以接受的。

在对假阳性和假阴性进行人工检查后，本章发现在字符差异很小但含义或语气不同的情况下，可能会发生假阴性的情况。例如，在本章的结果中，一个变异翻译有一个额外的 而，而这在原始翻译中不存在。本章人工检查认为这是不一致的，而本章所用的相似度指标则认为其是一致的。当两个翻译中有许多不同的词但它们具有相同的含义时，可能会发生假阳性。例如，在汉语中，尚未和还没有都表示“还没有”，但表达每个短语所用的字符完全不同。

测试过程中的假阳性可能带来的危害在于本章的方法可能会使翻译变得更差。本章在下文的有效性中探索了这种可能性。

表 5.6　针对不同翻译检测方法报告的问题的有效性

方法	TN	FN	FP	TP	精确率	召回率	F1
CAT-L	0.48	0.02	0.15	0.35	0.70	0.95	0.80
CAT-N	0.46	0.04	0.15	0.35	0.70	0.90	0.79

不一致问题的总数

确定阈值后，将 CAT-N 和 CAT-L 生成的每个检测输入输入到谷歌翻译和 Transformer 当中以获取其翻译，然后应用相似度指标及阈值来自动化确定检测输入的翻译

表 5.7　CAT 所检测出的不一致性问题数量 (RQ2)

	指标	CAT-L	CAT-N
GT	LCS	2,198	5,109
	ED	2,210	5,128
	TFIDF	2,430	5,381
	BLEU	2,126	4,852
Transformer	LCS	1,957	4,545
	ED	1,963	4,573
	TFIDF	2,146	4,798
	BLEU	1,897	4,325

是否存在不一致性问题（低于阈值）。

表 5.7 显示了 CAT-N 和 CAT-L 报告的每个相似性指标的问题数量。对于 CAT-L 而言，在 Transformer 模型上不一致的翻译测试结果个数分别是 LCS: 1,957 ； ED：1,963；tf-idf: 2,146 ； BLEU：1,897。因此，总体而言，大约五分之二的翻译低于一致性阈值。对于谷歌翻译，不一致的翻译结果个数分别是 LCS: 2,198；ED：2,210；tf-idf：2,430；BLEU：2,126。可以观察到 CAT-N 在报告的问题数量上优于 CAT-L。对于 CAT-N 而言，在 Transformer 模型上不一致的翻译测试结果个数分别是 LCS: 4,545 ； ED：4,573；tf-idf: 4,798 ；BLEU：4,325。对于谷歌翻译，不一致的翻译结果个数分别是 LCS: 5,109；ED：5,128；tf-idf：5,381；BLEU：4,852。平均而言，CAT-N 在谷歌翻译和 Transformer 上检测到的问题比 CAT-L 多 129%。

本章进一步调查报告不一致性问题的多样性。结果如图 5.16所示，这里以 LCS 相似度度量上报告的问题为例。可以观察到 CAT-N 报告了由 10 种类型的词性引起的各种问题。CAT-L 只能检测名词、形容词和数字的问题，而几乎无法检测到其他词性中的单词引起的问题。CAT-N 比 CAT-L 生成更多数量、更多样化和更广泛分布的变异体，这有助于其问题检测能力。

因为相似度度量作为检测结果可能与人类所给出的检测结果不同（即人类判断翻译是否一致）。本章统一从每种方法在 Transformer 模型的测试结果中采样 100 个检测用例①。对于每个采样的检测输入及其翻译结果，手动检查检测输入及其翻译是否存在不一致的问题②，然后将人工判断与相似度指标的判断进行比较。

本章展示了人工检查结果的精确率、召回率和 F1 分数。结果表明，对于 CAT-L，相似度的精度为 0.70，召回率为 0.95，F1 分数平均为 0.80。对于 CAT-N，相似性度量的精度为 0.72，召回率为 0.90，平均 F1 分数为 0.80。这些结果表明 CAT-N 报告的问题的有效性与 CAT-L 相似。CAT-L 的召回率略高于 CAT-N。这是因为 CAT-L 生成的许

① 本章以相等的概率对成功揭示问题的检测用例和未能揭示问题的检测用例进行抽样。
② 卡帕分数的平均值为 0.97 。

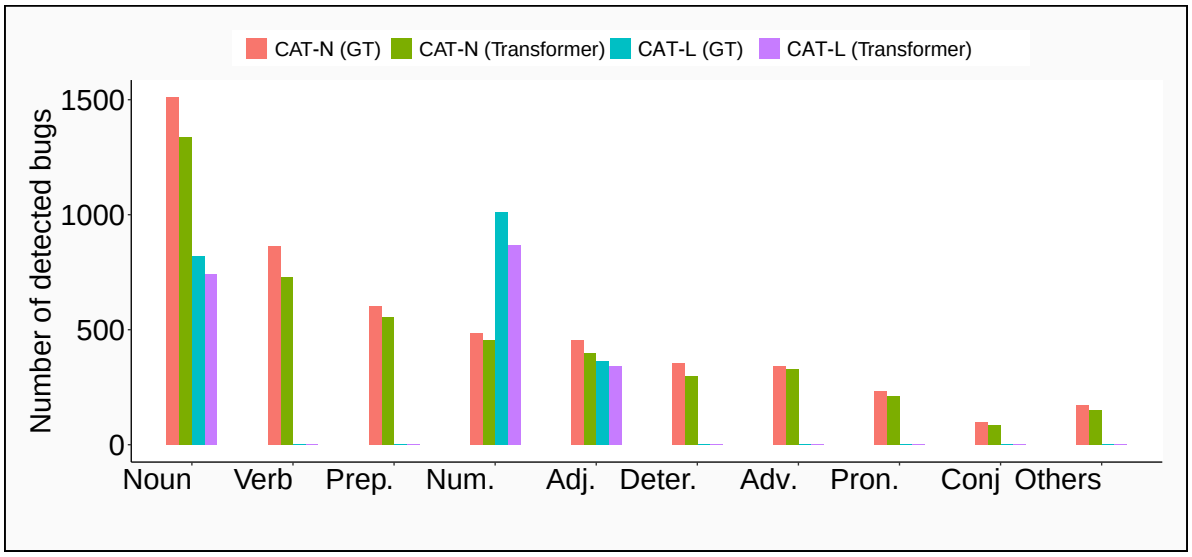

图 5.16　每个变异类型（x-axis）中，CAT 检测到的问题数量（y-axis）（此图显示 CAT-N 报告的问题比 CAT-L 更多样化）

多变异体是通过数字替换产生的（如图 5.13 所示），它对翻译一致性的影响小于其他变异体类型。

总的来说，人工检查表明 CAT-N 报告的问题的有效性类似于 CAT-L 报告的问题。

对 **RQ2** 的回答：在检测 Transformer 模型中的不一致性问题时，CAT-L 的 F1 分数为 0.80，CAT-N 和 CAT-L 具有类似的效果，其 F1 分数也为 0.80。CAT-L 可以检测到三种问题类型，在不同指标上可以检测到约 2,000 个不一致性问题。CAT-N 在多样性上检测到的问题类型比 CAT-L 多七种，并且在数量上比 CAT-L 多 129%。

CAT 在修复一致性问题方面的能力 (RQ3)　为了便于比较，本章让 CAT-N 和 CAT-L 修复同一组检测到的问题，即由 CAT-N 检测到的问题。对于每个问题，CAT-N 和 CAT-L 在原句和测试用例中的变异体上都生成最多 16 个变异体（第 5.2.2.1 节介绍的修复变异数上限）进行自动翻译修复. 对于 Transformer，本章使用交叉引用进行黑盒修复，同时也使用预测概率进行灰盒修复。谷歌翻译的预测概率是无法获得的，因此本章只进行黑盒修复。

为了回答 RQ3，本章先展示了通过相似度度量计算得到的已修复问题的数量。然后，本章展示了不同词性中的单词引起的修复问题的多样性。最后，本章展示了通过人工检查获得的翻译的修复能力。

通过相似性度量访问的修复效果

结果显示在表 5.8 中。每个单元格根据相似性指标显示已修复的问题数量，以及

已修复问题的比例（相对于 CAT-N 所检测到的问题总数）。上面几行是谷歌翻译 (GT)，下面几行是 Transformer 模型。

CAT-L 在谷歌翻译/Transformer 上以黑盒修复的形式可以修复平均 17%/15% 的不一致性问题，当使用了灰盒修复的方式，其效果获得了一定提升。可以观察到 CAT-N 修复的问题比 CAT-L 多。例如，对于具有 LCS 相似度指标的 Google 翻译，CAT-N 修复了 53% 的报告问题，而 CAT-L 仅修复了 17%。平均而言，CAT-N 在谷歌翻译/Transformer 上修复的问题比使用黑盒修复（使用交叉引用）的 CAT-L 多 199% / 238%，比在 Transformer 上使用灰盒修复的 CAT-L 多 190%（使用预测概率）。

表 5.8　修复不一致性问题的数量和比例 (RQ3)

方法	指标	基于预测概率	基于交叉引用
CAT-L (GT)	LCS	-	890 (17%)
	ED	-	890 (17%)
	TFIDF	-	980 (18%)
	BLEU	-	886 (18%)
CAT-N (GT)	LCS	-	2,729 (53%)
	ED	-	2,730 (53%)
	TFIDF	-	2,741 (51%)
	BLEU	-	2,719 (56%)
CAT-L (Transformer)	LCS	738 (16%)	684 (15%)
	ED	744 (17%)	689 (15%)
	TFIDF	810 (17%)	748 (16%)
	BLEU	739 (17%)	708 (16%)
CAT-N (Transformer)	LCS	2,193 (48%)	2,399 (53%)
	ED	2,196 (48%)	2,399 (52%)
	TFIDF	2,231 (47%)	2,415 (50%)
	BLEU	2,176 (50%)	2,344 (54%)

修复问题的多样性

本章进一步研究了 CAT-N 在不同类型的已修复问题中的有效性，这些问题由不同词性中的单词替换所检测到。由于不同类型有不同数量的报告问题，本章仅关注修复的问题与每种类型报告的问题数量的比例。

结果如图 5.17所示①。可以观察到 CAT-N 在所有问题类型中相比 CAT-L 都具有更好的效果。此外，可以发现 CAT-N 在所有问题类型中修复了 Transformer / 谷歌翻译上 42%-64% / 43%-68% 的问题。CAT-L 修复了数字中 45% / 50% 的问题，但仅修复了 Transformer / 谷歌翻译上其他问题类型中的 5%-14% / 7%-18% 的问题。CAT-L 为名词、

① 以 LCS 相似度度量上的基于交叉引用修复的不一致性问题比例为例

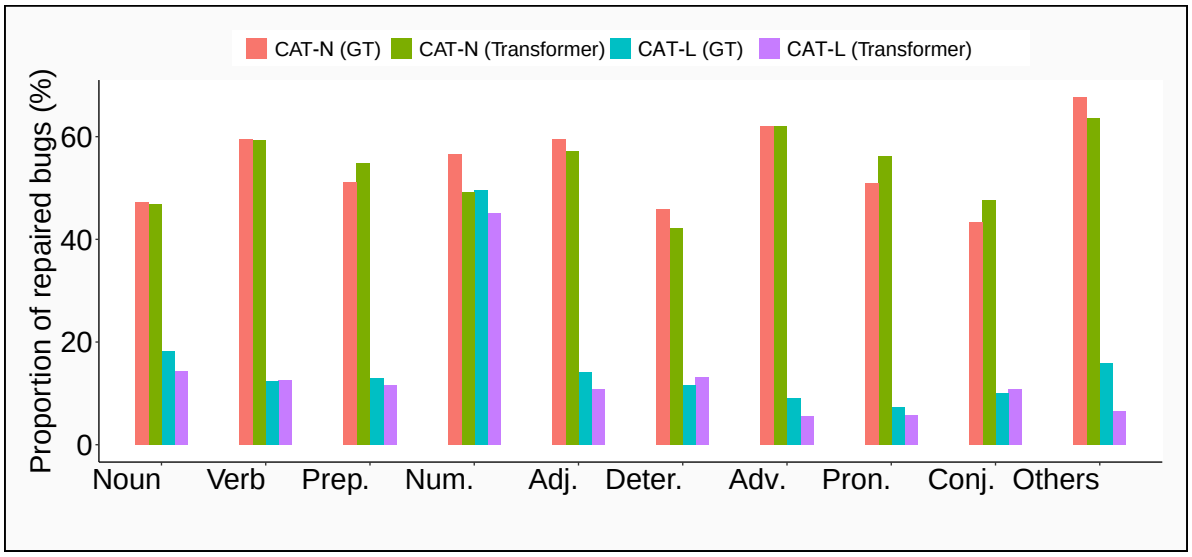

图 5.17　每个问题类型（x 轴）已修复的不一致性问题与所检测到的不一致性问题数的比例（y 轴）

形容词和数字生成变异体，但它仍然可以修复其他类型的问题。这是因为不同类型的替换产生的未更改部分的语义可能是相同的。

修复翻译的有效性

本章人工手动检查了 LCS 相似性指标上使用交叉引用的方式对 Transformer 模型的不一致性问题修复结果。目的是检查基于相似度度量的问题修复是否确实提高了翻译的一致性。

尽管 CAT-N 旨在修复翻译不一致，但它实际目的则是感知领域知识并提升翻译可接受性，它捕获了翻译满足人类对合理（也称为可接受）翻译的评估的属性。因此，人工检查考虑两个方面：1）修复前后翻译的一致性；2) 原句译文的可接受性以及修复前后的变体。对于每个维度，该实验设置了三个标签“提升”、“不变”和“下降”。本章随机抽样了 100 个报告的修复。对于可接受性而言，本章检查了原始句子和变异体的翻译改进，因此需要进行 200 次人工检查。

表 5.9 显示了实验结果①。从该表中，为了提高一致性，CAT-N 的修复成功地提高了 93 (93%) 个翻译的一致性，其余 7 (7%) 个翻译的一致性保持不变。而对于 CAT-L，它的修复提高了 87 (87%) 个翻译的一致性，12 个 (12%) 翻译的一致性保持不变，1 (1%) 个翻译的一致性有所下降。为了验证翻译的可接受性，CAT-N 提高了 32 个 (16.0%) 句子的翻译可接受性，降低了 15 (7.5%) 个句子的可接受性（由于可能在质量和一致性之间进行权衡以及词对齐工具问题所导致的问题修复）。CAT-L 提高了相同数量句子 (32 个) 的可接受性，但导致 19 (9.5%) 个句子的可接受性下降。

① 翻译可接受性和翻译一致性的卡帕指数分数分别为 0.96 和 0.97。

对 **RQ3** 的回答: CAT-L 的黑盒修复平均减少了 17%/15% 谷歌翻译/Transformer 的翻译一致性问题。其灰盒修复平均减少了 Transformer 17% 的翻译一致性问题。人工检查表明，修复后的翻译在 87% 的情况下提高了一致性（降低了 1%），在 32% 的情况下具有更好的翻译可接受性（9.5% 更差）。CAT-N 修复的问题是 CAT-L 的两倍。对于多样性而言，CAT-N 成功修复了每种问题类型下近一半的问题，而 CAT-L 仅在数词上实现了具有竞争力的性能（CAT-N 为 53%，而 CAT-L 为 47%），其未能修复其他类型下占有 89% 的翻译一致性问题。

表 5.9　关于问题修复的人工检查结果 (RQ3)

	角度	提升	不变	下降
CAT-L	**一致性**	87	12	1
	可接受性	32	149	19
CAT-N	**一致性**	93	7	0
	可接受性	32	153	15

5.2.2.3　扩展分析和讨论

本节对所提出的方法提供进一步的细节和分析。

翻译修复示例　表 5.10/表 5.11 给出了一些 CAT-L/CAT-N 模型修复误译的例子。第一列是翻译输入；第二列显示原始翻译输出（转换为拼音），其中斜体字解释误译部分；最后一列显示了由 CAT-L/CAT-N 所修复的翻译。

数据增强的有效性和效率　之前的工作已经采用数据增强来提高机器学习模型的鲁棒性[125,126]。在本章的工作中，对于已知源代码的开发人员来说，通过数据增强重新训练模型也是提高翻译一致性的候选解决方案。

表 5.10　CAT-L 模型修复的问题示例

输入	**原始翻译**	**修复后翻译**
Female students do good research in computer science.	女学生在计算机科学方面做了很多研究[问题: “good” → ‘‘很多’’。]	女学生在计算机科学方面做了很好的研究 [“good” 被正确翻译到 “很好的”。]
If you need help, you can enjoy timely services by pressing a nearby one of the 41 call buttons in the station.	如果你需要帮助，你可以通过按附近的 41 个呼叫按钮享受及时的服务。[问题: “one of” 并没有被翻译。]	如果你需要帮助，你可以通过按附近的 41 个呼叫按钮中的一个 来享受及时的服务。[“one of” 被正确翻译。]

表 5.11　CAT-N 模型修复的问题示例

输入	原始翻译	修复后翻译
Around 140,000 people have a heart attack in England every year, and a quarter of these go on to have another attack or a stroke.	每年在英国约有140 万 人心脏病发作，其中四分之一继续发作或中风。[问题: “140,000” 被问题翻译成 “140 万’’。]	每年在英国约有140,000 人心脏病发作，其中四分之一继续发作或中风。[“140,000” 正确翻译到 “140,000’’。]
Your patience never fails to disappoint me.	你的耐心永远不会 让我失望 [问题: “never fails to” 被问题翻译到其具有相反意思的词语 ‘‘永远不会’’。]	你的耐心总是 让我失望 [“never fails” 被正确翻译成 ‘‘总是’’。]
There he managed presidential protocol and government staff, the Kremlin website says (in Russian).	克里姆林宫网站说，他在那里管理着总统协议和政府工作人员。[问题: “in Russian” 并没有被翻译。]	那里，他管理了总统协议和政府工作人员，克里姆林宫网站说（俄文）。[“in Russian” 被正确翻译到 ‘‘俄文’’。]

为了研究这个方案，本章设计了实验来研究添加更多的训练数据是否会产生更好的翻译一致性。本章控制了训练数据的大小，分别使用了原始训练数据的 10%、20%、...、90% 来构建 Transformer。图 5.18 显示了结果。当训练数据比率的大小在 0.7 和 1.0 之间时，没有观察到翻译不一致性存在下降趋势。这表明增加训练数据在改善翻译不一致方面的效果可能有限。

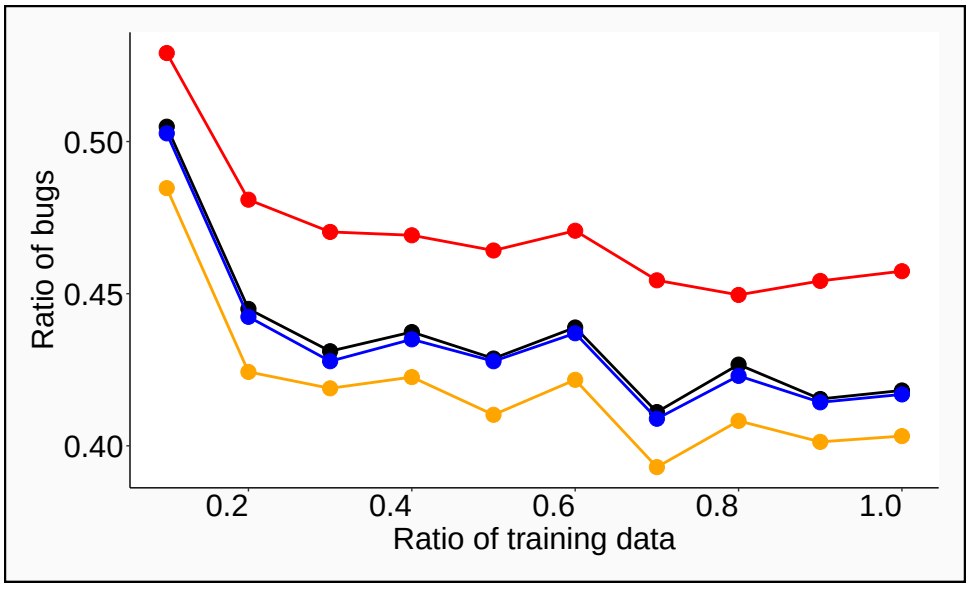

图 5.18　在 Transformer 模型下不同训练数据大小（x 轴）对应不一致问题的比率（y 轴）

同时，数据增强需要模型再训练。可以发现，在当前的实验使用单个显卡的配置下，使用 100% 的训练数据来训练翻译模型需要多达 19 个小时（更多详细信息请参见章节 5.2.2）。在实践中，数据收集、标记和处理也需要大量的额外人工时间。总之，几乎没有证据可以表明数据增强是一个修复翻译不一致性问题的完美解决方案。

与训练数据增强等模型再训练方法相比，CAT 具有以下优势：**1)** CAT 既不需要源

代码也不需要训练数据，要么是完全黑盒，要么只需要预测概率（灰盒）；**2)** CAT 可以有更低的修复成本，因为它不需要额外的数据，也不需要模型再训练；**3)** CAT 更灵活，因为它可以修复特定的问题，而无需影响其他由翻译系统生成的良好翻译。

CAT 模型的效率　本章记录了 CAT-N 和 CAT-L 在变异体生成、问题检测和问题修复过程中的所需要的时间成本。结果如表 5.12所示。总体而言，在本章的配置下，CAT-N 和 CAT-L 在它们的自动检测和修复任务中都有很好的效率（更多细节请参见章节 5.2.2.1）。这种高效率非常重要，因为用户在线使用机器翻译时不应该等待过久的时间。

具体来说，对于变异体生成，CAT-N 的平均时间成本为每个变异体 0.01 秒，CAT-L 的平均时间成本为 0.41 秒。原因是 CAT-L 使用 Stanford 解析器来解析每个句子以应用单词替换以及过滤变异候选者。CAT-N 直接使用了 BERT 模型的推理，它避免了解析过程，从而大大提高了效率，但是依赖于高性能显卡。

表 5.12　CAT-N 和 CAT-L 模型的效率 (RQ4)

方法	CAT-L	CAT-N
变异体生成	0.41s	0.01s
问题检测	0.99s	0.27s
问题修复	1.34s	1.92s

CAT-N 在变异生成中的成本较低也有助于其在问题检测方面的成本较低：CAT-N 平均需要 0.27 秒来检查每个句子以报告问题，而 CAT-L 需要 0.99 秒。

对于每个报告的问题，CAT-N 花费 1.92 秒进行自动修复，CAT-L 花费 1.34 秒。原因是 CAT-L 在修复问题时无法生成许多变异体，但每个变异体都需要从谷歌/Transformer 获取翻译以进行修复过程。因此，它花费在每个问题上的时间低于 CAT-N。然而，这种较短的时间需要付出巨大的代价：由于变异体的不足，大多数问题无法成功修复（正如本章在 RQ2 中观察到的那样）。

变异数量设置的影响　本章将用于不一致性问题检测的最大变异体数设置为 5 ，将用于不一致性问题修复最大变异体数设置为 16 。在这里，本章研究了最大变异体数量设置对检测和修复的问题数量的影响。

对于问题检测，本章重复了 RQ2 的实验，并设置每个句子最多生成 1 或 3 个变异体。表 5.13 显示了在四个指标上使用不同最大变异体数量时由 CAT-N 和 CAT-L 检测到的不一致性问题数量。本章说明更多的变异体会使得 CAT-N 和 CAT-L 检测到更多的

表 5.13　最大变异体数对不一致性问题检测的影响

指标	CAT-L			CAT-N		
	1	3	5	1	3	5
LCS	575	1,385	1,957	1,032	2,707	4,545
ED	576	1,389	1,963	1,036	2,728	4,573
TFIDF	627	1,511	2,146	1,049	2,861	4,798
BLEU	553	1,340	1,897	1,001	2,577	4,325

问题。值得注意的是，即使对于三个不同的设置，CAT-N 检测到的问题大约是 CAT-L 检测到的问题的两倍。

表 5.14　最大变异体数对不一致性问题修复的影响

指标	CAT-L			CAT-N		
	4	8	16	4	8	16
LCS	516	605	684	1,724	2,117	2,399
ED	526	611	689	1,728	2,141	2,399
TFIDF	584	678	748	1,747	2,139	2,415
BLEU	536	635	708	1,667	2,053	2,344

对于一致性问题修复，本章使用最多 4 或 8 个变异体重复 RQ3 的实验。表 5.14 记录了相关结果。可以看出，无论使用什么上限，CAT-N 修复的问题数量一直是 CAT-L 修复的三倍以上。有趣的是，可以观察到即使有 4 个变异体，CAT-N 仍然可以修复比使用 16 个变异体的 CAT-L 多 143% 的问题。这是因为 CAT-L 无法为许多句子生成变异体（正如从图 5.12 中观察到的一样），因此它难以修复由这些情况导致的任何问题。

与其他翻译检测方法的比较　现有的工作中还有一些其他的翻译检测方法。本章将在本节探讨它们的有效性并讨论它们与 CAT 的区别。在这类工作当中，不同的工作会使用不同的扰动约束。根据扰动约束来分类，其主要包括：1）基于交叉引用的检测模型；和 2）基于输入生成的检测模型。本章先对其方法进行描述。

基于交叉引用的检测模型

Pesu et al.（2018）[127]提出了一种基于交叉引用的检测方法。该方法假定同一句子的直接翻译（从源语言到目标语言）和间接翻译（从源语言到中间语言，然后从中间语言到目标语言）应该相同。否则的话，则将该方法将其汇报为一个错误。Cao et al.（2020）[128]则提出了类似的想法，其通过一个检测输入去检测不同翻译系统之间的翻译差别，其中，一个句子在多个翻译器上预期具有相似的翻译。

基于输入生成的检测模型

主流方法则通过输入和生成的变异体输入之间的关联关系来对机器翻译系统进行检测[112]。他们采用在输入的源语言中，对输入进行变异修改从而生成新句子或短语作为输入的策略。具体来说，给定源语言中的一个输入句子，该类方法会生成与其一个相关的句子或短语，然后将原始的输入句子和其生成的句子（短语）输入到待测机器翻译系统当中。进而，该方法将两者的翻译结果进行比较，以检测是否发现了错误。特别的是，Purity[129]将输入句子分解为短语，并假设每个单独的短语（无上下文）的翻译应与整个句子所提供的上下文中该短语的翻译相似。

与 Purity 不相同的是，其他基于输入生成的测试模型的方法都采用单词替换的方式用以生成新的输入。换句话说，这些方法通过将原句中的一个词替换为另一个相关词来生成一个新句子作为输入。Gupta et al.（2020）[130] 通过将一个单词替换为另一个具有完全不同含义的单词来检测翻译错误，并期望被替换的单词应和原单词具有不同的翻译。Sun 和 Zhou（2018）[131]）通过在 ‘‘喜欢’’ 或 ‘‘讨厌’’ 之前的人名替换生成检测输入，他们认为转换前后句子的翻译应该相似。Gupta et al.（2020）[130] 通过将一个单词转换为另一个含义完全不同的单词来检测翻译错误，并期望该转换可以产生不同的翻译。He et al.（2020）[58]在输入句子中的单词转换为另一个单词的同时保持句子结构的不变，假设输入句子和转换所得句子的翻译也应具有相似的结构。

CAT 和现有机器翻译检测方法的比较

本章进一步将 CAT（基于 LCS 指标）与最近发布的三种检测方法进行比较：SIT[58]、PatInv[130] 和 Purity[132]。对于 SIT、PatInv 和 Purity，本章使用他们发布的代码并根据如他们的论文所示的最佳性能参数对它们进行参数调整。表 5.15 展示了这四种方法在谷歌翻译 (GT) 和 Transformer 上检测到的问题数量。从这些结果可以看出，CAT 和 SIT 检测到的问题比 CAT-L、PatInv 和 Purity 多。

表 5.15　不同机器翻译检测方法所检测到的问题数量

翻译系统	SIT	PatInv	Purity	CAT-N	CAT-L
GT	5,576	82	3,459	5,109	2,198
Transformer	5,368	99	3,518	4,545	1,957

每种方法定义了属于自己的扰动约束并根据该扰动约束报告所检测到问题。因此，它们具有各种问题类型，其问题不一定是本章所研究的一致性问题。为了探索将这些方法用于与 CAT-N 实现相同的目的（即检测一致性问题）的可能性，本章进一步对每种方法所报告的问题进行人工手动检查，并报告它们在检测不一致性问题中的有效性。

为此，本章手动检查了[①] 报告的问题，并检查它们是否真的揭示了翻译不一致的问题。这个分析的结果记录在表 5.6 中。在此表中，假阳性 (FP) 意味着该方法将翻译报告为有问题，但手动检查实际上发现它是正确的，即它不是问题。假阴性 (FN) 意味着该方法将翻译报告为没有问题，但手动检查显示它实际上是有问题的（即存在问题）。可以观察到 CAT-N 的 F1 分数高于 SIT、PatInv 和 Purity。考虑到 SIT、PatInv 和 Purity 旨在检测不同类型的问题，这样的效果并不令人奇怪。

5.3 小结

本文所提出的技术本质上用于解决约束感知问题，而其中命名约束感知问题和扰动约束感知问题也存在于无属性图分类任务和机器翻译任务上。因而，本章将所提出的约束感知算法拓展到这两个领域。

详细来说，本章将迭代筛选算法推广至无属性图分类领域，并与图神经网络技术相结合。在布尔表达式求解任务和最大独立集求解任务的实验结果表明，迭代筛选算法可以使得预测错误率相比于各自任务对应的现有最佳方法下降了 76%（布尔表达式求解）和 39%（最大独立集求解）。

同时，本章将检测和修复算法（CAT）推广应用至机器翻译任务之上。本章分别在一个工业级机器翻译系统（谷歌翻译）和一个学术界常用机器翻译系统（Transformer）上进行验证。其实验结果表明，CAT 在生成具有一致翻译的输入时具有很高的精度 (96.5%)。同时，CAT 发现在 Transformer 模型上大约有 40% 的不一致性问题。黑盒修复平均为谷歌翻译和 Transformer 模型修复了 53% 和 52% 的问题。灰盒修复平均为 Transformer 修复了 48% 的问题。人工检查表明，通过 CAT 所修复的翻译在 93% 的情况下都提高了翻译一致性，并且 CAT 的修复在 16% 的情况下具有更好的翻译可接受性（7.5% 则更差）。

本章的拓展应用方法说明了本文所提方法的通用性及有效性。

① 人工检查 SIT / PatInv / Purity 的卡帕指数分别为 0.94 / 0.89 / 0.92

第六章　总结和展望

6.1　总结

程序是现代社会信息化发展必不可少的元素。由开发者开发程序往往需要耗费大量开发时间。如何有效地提高开发者开发程序代码的效率，减轻开发者的开发负担一直是代码相关研究领域所关注的问题。因而，程序生成日益显示出其研究的重要性。

本文工作主要围绕程序生成中的程序语法约束、语义约束、任务特定的约束三类约束的感知问题，提出相应的感知方法。主要目标是提升在程序生成任务下的神经网络模型感知约束能力，从而进一步提升程序生成任务的效果，辅助开发者进行开发。

对于程序语法约束感知，本文针对抽象语法树知识，设计了一种基于 Transformer 模型的程序语法约束感知方法（TreeGen，第二章）。该模型针对程序生成任务中的语法规则结构编码问题，提出了一种树形结构网络。从而使得程序生成模型拥有了较强的语法约束结构的编码能力。同时针对语法约束编码中另一个长程依赖问题，本文使用了在自然语言处理任务中以缓解长程依赖问题著称的 Transformer 模型，有效地缓解了语法约束中长程依赖问题，提升了程序生成的效果。

对于语义约束感知，本文针对变量、函数名命名约束感知问题，分析了现有工作的不足，提出一种基于迭代筛选的命名约束感知方法（迭代筛选算法，第三章）。

对于任务特定约束的感知，本文针对一种程序生成模型中的扰动约束感知问题提出了基于检测和修复的扰动约束感知方法（检测和修复算法，CAT，第四章）。

总的来说，在同时处理以上三种约束时，对于一个程序生成模型，本文先使用 TreeGen 作为基础模型来解决程序语法约束感知问题。在 TreeGen 模型的基础上，模型的训练使用迭代筛选算法进一步解决语义约束问题。在模型预测时则使用了 CAT 算法来解决任务特定的约束感知问题。

最后，本文所研究的命名约束感知问题和扰动约束感知问题并不局限于程序生成，其存在于机器学习中的诸多任务中。因而，本文进一步将所提出的命名约束感知方法应用于需要命名约束感知的无属性图分类任务。同时，本文也将扰动约束感知方法拓展应用到需要扰动约束感知的机器翻译任务中（第五章）。在拓展应用之后，本文通过实验验证了本文所提出的约束感知方法的有效性和通用性。

- **基于 Transformer 模型的程序语法约束感知** 现有的程序生成工作难以很好地感知程序语法约束，其神经网络架构依赖于循环神经网络。该类网络是一种用以表示一串序列的“扁平”化神经网络，其难以感知到抽象语法树本身含有的语法树结构信息。同时，循环神经网络技术也会受制于神经网络本身所含有的长

程依赖问题的影响，从而难以解决程序中常见的长程依赖问题。

本文提出了一种新颖的神经网络结构 TreeGen 用于程序生成。为了解决语法规则中的长程依赖问题，TreeGen 采用了最新提出的 Transformer 神经网络结构，该神经网络结构能够有效捕获长程依赖关系。但是，原始的 Transformer 体系结构不是为程序设计的，并且不能利用树结构。为了解决语法结构编码问题，TreeGen 使用了一种新的抽象语法树阅读器。

总的来说，TreeGen 神经网络结构由三部分组成：（1）自然语言阅读器（NL Reader）（编码器）用以对输入文本描述进行编码；（2）抽象语法树阅读器（AST Reader）（前几个 Transformer 解码器模块）使用基于树结构的卷积子层对所生成的部分语法规则序列进行编码；（3）解码器（Decoder）（其余的 Transformer 解码器模块）将 Query 信息（下一个待扩展节点的位置信息）与前两个编码器组合在一起，以预测下一个要使用的语法规则。

该方法在 Python 数据集和语义解析数据集上进行验证，其实验结果说明 TreeGen 在 Python 生成任务上的表现比之前的方法高出 9 个百分点，而在语义解析数据集上也取得了基于神经网络的方法中最好的生成准确率。

- **基于集成学习的命名约束感知** 对于语义约束感知问题，本文针对变量、函数命名命名约束进行研究。该命名约束表示：变量、函数名具有可以在不改变程序语义的情况下可以任意重命名的性质。现有的工作关于变量、函数名编码的工作存在难以感知该约束的问题，进而使得神经网络难以适应具有不同的变量、函数名命名风格或是命名质量较低的数据。

  为了感知变量、函数名的命名约束，本文先提出了一种基于迭代筛选的命名约束感知方法。该算法首先通过随机生成的方式在不影响语义的情况下对变量、函数名进行随机交换，并使用交换后的数据进行训练。该方法由于随机命名的原因，可以让程序生成方法适应不同的变量、函数命名风格。

  然而，以随机生成的方式所训练的神经网络会受到随机命名风格的局限性的影响。为了解决这个问题，本文进一步提出了一种迭代筛选算法，该技术通过一种无监督的方式让神经网络学会如何为变量、函数名分配合适的名称以避免了上述局限性，进而提升了程序输出的效果。

  在 Github 所爬取的数据集上的实验结果表明，本文所提出的迭代筛选算法相比于不使用迭代筛选算法的技术生成程序的准确率提高了 8 个百分点。
- **基于集成学习的扰动约束感知** 程序生成是一种从输入自然语言描述输出到目标程序语言代码的任务。现有的程序生成工作忽视了输入输出之间理应满足的扰动约束，从而对程序生成系统的生成所生成程序的可接受性造成了影响。

为了提升现有程序生成工作所生成程序的可接受性，本文引入了一种基于检测和修复的扰动约束感知方法（CAT）。该方法对输入生成语句进行上下文相似转换，以生成其对应变异体句，可用作被测程序生成系统的自然语言输入。当上下文相似转换的未变异部分对应的程序具有较大的修改时，该方法将报告其生成的程序具有约束不一致性问题。

针对所检测出的约束不一致性问题，CAT 进一步通过一种基于集成学习的技术实现轻量级的黑盒修复技术以修复约束不一致性问题，进而提升程序生成系统所生成程序的可接受性。CAT 是第一种以纯黑盒方式提升程序生成系统的修复技术。在引入黑盒修复的同时，CAT 还引入了一种灰盒修复的方法，以提升程序生成系统所生成程序的可接受性。

本文在 CodeGPT 这个常用的程序生成工具上对 CAT 进行评估。CAT 的自动检测方法发现：在 CodeGPT 模型上，大约有 39% 的输入存在约束不一致性问题。而 CAT 的黑盒修复平均为 CodeGPT 修复了 42% 的问题。灰盒修复平均为 CodeGPT 修复了 33% 的问题。该实验表明，CAT 使得程序生成方法感知到了所定义的扰动约束，进而达到提升程序生成方法所生成程序的可接受性的目的。同时，进一步的人工检查表明，通过本文的方法所修复的问题在 90% 的情况下也提高了生成稳定性。

- **迭代筛选算法和检测和修复算法的拓展应用** 本文将所提出的迭代筛选算法及基于集成学习的扰动约束感知方法 CAT 进一步进行拓展，将其应用于具有相近约束感知问题的任务上。

  本文将迭代筛选算法应用于无属性图节点分类中的布尔表达式求解和最大独立集求解问题上。本文所提出方法相比于之前效果最佳的方法预测错误数量下降了 76%（布尔表达式求解）和 39%（最大独立集求解）。本文将检测和修复算法（CAT）拓展应用于机器翻译任务上，CAT 的验证使用了自动化的一致性指标，同时本文也使用了人工评估的方法。本文发现在 Transformer 模型上大约有 40% 的翻译存在不一致性问题。黑盒修复平均为谷歌翻译和 Transformer 模型修复了 53% 和 52% 的问题。灰盒修复平均为 Transformer 修复了 48% 的问题。最后，从提升翻译效果的角度，CAT 的修复在 16% 的情况下都具有更好的翻译可接受性（7.5% 更差）。

综上所述，本文从程序生成中的程序语法约束、语义约束、任务特定的约束三类约束的感知问题，提出其相应的感知方法，并分别设计实验进行验证。同时，上述约束感知问题并不局限于程序生成，该问题也存在于其他领域中。因而，本文将所提出的约束感知方法拓展应用于无属性图分类和机器翻译任务上当中，体现了本文所提出

方法的有效性和通用性。

6.2 未来研究工作

基于本文现有的研究，本文的未来研究工作可以从神经网络编码及表示程序的角度出发，借助大规模的开源数据和神经网络技术，提出一种更加有效的程序生成方法和一种通用的程序表示方法。

使用神经网络编码及表示程序仍旧面临着诸多挑战：

- **数据量不足**

  随着大规模开源时代的到来，尽管常用的编程语言（如 C/C++，Java，Python 等）近乎坐拥取之不尽的开源代码数据，对于部分小众编程语言（如.Net 平台下的诸多语言）来说，代码数据依旧是一种稀缺的资源。而现有的神经网络技术无法仅仅通过少量数据完成对程序编码的学习，这会使得现有的程序相关的技术（如程序生成方法，程序补全技术，注释生成技术）无法使用在这类语言之上，显著提升了开发的复杂度。针对这类问题，根据本人已有的工作及研究过程，本人发现不同编程语言之间其实具有相通性，而这种相通性更多的表示在程序的抽象语法树之上，相近的程序往往具有相近的抽象语法树。通过多种语言协同训练的方法，可以使得数据量较小的语言通过程序抽象语法树的相似性从数据量较多的程序语言中汲取约束。这种方法可以提升其程序编码的效果并使得小众语言使用现有的神经网络技术成为可能。

- **无法编码动态执行信息**

  现有的程序编码模型大多依赖于静态的程序信息，而对于一个程序来说，静态信息只是一小部分，程序中的语义往往需要动态执行才能具体表现出来。针对这类问题，本人的初步思路是通过符号执行技术去完成静态程序的执行，然后将符号执行的数据输入到神经网络之中，从而使得神经网络可以在拥有动态信息的基础上拥有动态的执行信息，帮助神经网络理解及编码程序。

- **调用信息编码难度较大**

  程序的函数与函数之间往往存在相互之间的调用关系。现有的神经网络编码工作直接通过函数名等信息对函数的调用进行编码，而没有将调用函数的函数体内容考虑在内。这会导致其在编码程序时无法很好的理解所调用函数对应信息，从而给程序表示带来一定编码缺陷。针对该问题，本人计划通过一种动静态相结合的训练方法，将函数的函数体内容的表示向量通过神经网络技术先摘要出来，然后进一步利用摘要的向量作为所调用函数的表示去辅助训练，从而使得神经网络可以通过所调用函数的函数体学到函数的具体内容，提升编码程序的

能力。

- **语义约束的感知**

  本文仅关注了语义约束当中的标识符命名约束。然而除开标识符命名约束之外，语义约束中也包含有其他约束（如：不同的程序写法可能对应相同的程序语义）。在这个方向上，本人计划进一步分析语义约束，并提取出其中最关键的语义约束。在此基础上，进一步提出一种约束感知方法来处理这类约束。

参考文献

[1] A. Krizhevsky, I. Sutskever, G. E. Hinton. “ImageNet classification with deep convolutional neural networks”. In: *Commun. ACM* 60.6 (2017): pp. 84-90. URL: http://doi.acm.org/10.1145/3065386.

[2] L. Deng, J. Li, J. Huang, et al. “Recent advances in deep learning for speech research at Microsoft”. In: *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*. IEEE, 2013: pp. 8604-8608. URL: https://doi.org/10.1109/ICASSP.2013.6639345.

[3] A. Bordes, X. Glorot, J. Weston, et al. “Joint Learning of Words and Meaning Representations for Open-Text Semantic Parsing”. In: *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2012, La Palma, Canary Islands, Spain, April 21-23, 2012*. Ed. by N. D. Lawrence, M. A. Girolami. Vol. 22. JMLR Proceedings. JMLR.org, 2012: pp. 127-135. URL: http://proceedings.mlr.press/v22/bordes12.html.

[4] J. Devlin, J. Uesato, S. Bhupatiraju, et al. “RobustFill: Neural Program Learning under Noisy I/O”. In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. Ed. by D. Precup, Y. W. Teh. Vol. 70. Proceedings of Machine Learning Research. PMLR, 2017: pp. 990-998. URL: http://proceedings.mlr.press/v70/devlin17a.html.

[5] X. Hu, G. Li, X. Xia, et al. “Deep code comment generation with hybrid lexical and syntactical information”. In: *Empir. Softw. Eng.* 25.3 (2020): pp. 2179-2217. URL: https://doi.org/10.1007/s10664-019-09730-9.

[6] W. Ling, P. Blunsom, E. Grefenstette, et al. “Latent Predictor Networks for Code Generation”. In: *ACL*. 2016: pp. 599-609.

[7] V. Raychev, P. Bielik, M. T. Vechev. “Probabilistic model for code with decision trees”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. Ed. by E. Visser, Y. Smaragdakis. ACM, 2016: pp. 731-747. URL: https://doi.org/10.1145/2983990.2984041.

[8] P. Bielik, V. Raychev, M. T. Vechev. “Program Synthesis for Character Level Language Modeling”. In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL: https://openreview.net/forum?id=ry%5C_sjFqgx.

[9] M. Rabinovich, M. Stern, D. Klein. “Abstract Syntax Networks for Code Generation and Semantic Parsing”. In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*. Ed. by R. Barzilay, M. Kan. Association for Computational Linguistics, 2017: pp. 1139-1149. URL: https://doi.org/10.18653/v1/P17-1105.

[10] L. Dong, M. Lapata. “Language to Logical Form with Neural Attention”. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. The Association for Computer Linguistics, 2016. URL: https://doi.org/10.18653/v1/p16-1004.

[11] L. Dong, M. Lapata. “Coarse-to-Fine Decoding for Neural Semantic Parsing”. In: *ACL*. 2018: pp. 731-742.

[12] P. Yin, G. Neubig. “A Syntactic Neural Model for General-Purpose Code Generation”. In: *ACL*. 2017: pp. 440-450.

[13] P. Yin, C. Zhou, J. He, et al. “StructVAE: Tree-structured Latent Variable Models for Semi-supervised Semantic Parsing”. In: *ACL*. ACL, 2018: pp. 754-765.

[14] P. Yin, G. Neubig. “TRANX: A Transition-based Neural Abstract Syntax Parser for Semantic Parsing and Code Generation”. In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 2018: pp. 7-12.

[15] H. Jiang, L. Song, Y. Ge, et al. “An AST Structure Enhanced Decoder for Code Generation”. In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 30 (2021): pp. 468-476.

[16] B. Xie, J. Su, Y. Ge, et al. “Improving Tree-Structured Decoder Training for Code Generation via Mutual Learning”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 2021: pp. 14121-14128.

[17] H. Jiang, C. Zhou, F. Meng, et al. “Exploring dynamic selection of branch expansion orders for code generation”. In: *arXiv preprint arXiv:2106.00261* (2021).

[18] Y. Xiong, B. Wang. “L2S: A framework for synthesizing the most probable program under a specification”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31.3 (2022): pp. 1-45.

[19] A. T. Nguyen, T. N. Nguyen. “Graph-Based Statistical Language Model for Code”. In: *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. Ed. by A. Bertolino, G. Canfora, S. G. Elbaum. IEEE Computer Society, 2015: pp. 858-868. URL: https://doi.org/10.1109/ICSE.2015.336.

[20] X. Gu, H. Zhang, D. Zhang, et al. “Deep API learning”. In: *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. Ed. by T. Zimmermann, J. Cleland-Huang, Z. Su. ACM, 2016: pp. 631-642. URL: https://doi.org/10.1145/2950290.2950334.

[21] X. Gu, H. Zhang, D. Zhang, et al. “DeepAM: Migrate APIs with Multi-modal Sequence to Sequence Learning”. In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*. Ed. by C. Sierra. ijcai.org, 2017: pp. 3675-3681. URL: https://doi.org/10.24963/ijcai.2017/514.

[22] M. Allamanis, M. Brockschmidt, M. Khademi. “Learning to Represent Programs with Graphs”. In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL: https://openreview.net/forum?id=BJOFETxR-.

[23] C. Lyu, R. Wang, H. Zhang, et al. “Embedding API dependency graph for neural code generation”. In: *Empirical Software Engineering* 26.4 (2021): pp. 1-51.

[24] Z. Yao, J. R. Peddamail, H. Sun. “Coacor: Code annotation for code retrieval with reinforcement learning”. In: *The World Wide Web Conference*. 2019: pp. 2203-2214.

[25] Q. Zhu, Z. Sun, Y a. Xiao, et al. “A syntax-guided edit decoder for neural program repair”. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2021: pp. 341-353.

[26] U. Alon, M. Zilberstein, O. Levy, et al. “code2vec: Learning distributed representations of code”. In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019): pp. 1-29.

[27] U. Alon, S. Brody, O. Levy, et al. “code2seq: Generating Sequences from Structured Representations of Code”. In: *International Conference on Learning Representations*. 2018.

[28] U. Alon, R. Sadaka, O. Levy, et al. “Structural language models of code”. In: *International conference on machine learning*. 2020: pp. 245-256.

[29] Q. Zhu, Z. Sun, X. Liang, et al. “OCoR: an overlapping-aware code retriever”. In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 2020: pp. 883-894.

[30] L. Mou, G. Li, L. Zhang, et al. “Convolutional Neural Networks over Tree Structures for Programming Language Processing.” In: *AAAI*. 2016: pp. 1287-1293.

[31] J. Li, Y. Wang, M. R. Lyu, et al. “Code Completion with Neural Attention and Pointer Networks”. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. Ed. by J. Lang. ijcai.org, 2018: pp. 4159-4165. URL: https://doi.org/10.24963/ijcai.2018/578.

[32] T. Ben-Nun, A. S. Jakobovits, T. Hoefler. “Neural code comprehension: A learnable representation of code semantics”. In: *Advances in Neural Information Processing Systems* 31 (2018).

[33] P. Fernandes, M. Allamanis, M. Brockschmidt. “Structured Neural Summarization”. In: *CoRR* abs/1811.01824 (2018). arXiv: 1811.01824. URL: http://arxiv.org/abs/1811.01824.

[34] B. Wei, G. Li, X. Xia, et al. “Code generation as a dual task of code summarization”. In: *Advances in neural information processing systems* 32 (2019).

[35] J. Zhang, X. Wang, H. Zhang, et al. “A novel neural source code representation based on abstract syntax tree”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019: pp. 783-794.

[36] X. Jiang, Z. Zheng, C. Lyu, et al. “TreeBERT: A tree-based pre-trained model for programming language”. In: *Uncertainty in Artificial Intelligence*. 2021: pp. 54-63.

[37] Z. Feng, D. Guo, D. Tang, et al. “CodeBERT: A Pre-Trained Model for Programming and Natural Languages”. In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. 2020: pp. 1536-1547.

[38] 曹英魁, 孙泽宇, 邹艳珍, 等. 一种结构信息增强的代码修改自动转换方法[J]. 软件学报, 2021, 32(4): 1006-1022.

[39] Y. Sui, X. Cheng, G. Zhang, et al. “Flow2Vec: value-flow-based precise code embedding”. In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020): pp. 1-27.

[40] F. Zhang, B. Chen, R. Li, et al. “A hybrid code representation learning approach for predicting method names”. In: *Journal of Systems and Software* 180 (2021): p. 111011.

[41] R. Wu, S. Yan, Y. Shan, et al. “Deep image: Scaling up image recognition”. In: *arXiv preprint arXiv:1501.02876* 7.8 (2015).

[42] A. Mikoajczyk, M. Grochowski. “Data augmentation for improving deep learning in image classification problem”. In: *2018 international interdisciplinary PhD workshop (IIPhDW)*. 2018: pp. 117-122.

[43] Z. Zhong, L. Zheng, G. Kang, et al. “Random erasing data augmentation”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 34. 2020: pp. 13001-13008.

[44] C. Shorten, T. M. Khoshgoftaar. “A survey on image data augmentation for deep learning”. In: *Journal of big data* 6.1 (2019): pp. 1-48.

[45] X. Zhang, J. Zhao, Y. LeCun. “Character-level convolutional networks for text classification”. In: *Advances in neural information processing systems* 28 (2015).

[46] J. Wei, K. Zou. “Eda: Easy data augmentation techniques for boosting performance on text classification tasks”. In: *arXiv preprint arXiv:1901.11196* (2019).

[47] S. Y. Feng, A. W. Li, J. Hoey. “Keep calm and switch on! preserving sentiment and fluency in semantic text exchange”. In: *arXiv preprint arXiv:1909.00088* (2019).

[48] F. Nesti, A. Biondi, G. Buttazzo. “Detecting adversarial examples by input transformations, defense perturbations, and voting”. In: *IEEE Transactions on Neural Networks and Learning Systems* (2021).

[49] G. Dong, J. Wang, J. Sun, et al. “Repairing Adversarial Texts through Perturbation”. In: *arXiv preprint arXiv:2201.02504* (2021).

[50] G. Wang, J. Sun, J. Ma, et al. “Sentiment classification: The contribution of ensemble learning”. In: *Decision support systems* 57 (2014): pp. 77-93.

[51] T. Chalothom, J. Ellman. “Simple approaches of sentiment analysis via ensemble learning”. In: *information science and applications*. Springer, 2015: pp. 631-639.

[52] Z. Li, Q. Chen, V. Koltun. “Combinatorial optimization with graph convolutional networks and guided tree search”. In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 2018: pp. 537-546.

[53] W. Zhang, Z. Sun, Q. Zhu, et al. “NLocalSAT: Boosting Local Search with Solution Prediction”. In: *IJCAI*. 2020: pp. 1177-1183. URL: https://doi.org/10.24963/ijcai.2020/164.

[54] L. Dong, M. Lapata. “Language to Logical Form with Neural Attention”. In: *ACL*. 2016: pp. 33-43.

[55] M. Rabinovich, M. Stern, D. Klein. “Abstract Syntax Networks for Code Generation and Semantic Parsing”. In: *ACL*. 2017: pp. 1139-1149.

[56] S. A. Hayati, R. Olivier, P. Avvaru, et al. “Retrieval-Based Neural Code Generation”. In: *EMNLP*. 2018: pp. 925-930.

[57] Y. Bengio, P. Simard, P. Frasconi. “Learning long-term dependencies with gradient descent is difficult”. In: *IEEE Trans. Neural Networks* 5.2 (1994): pp. 157-166.

[58] P. He, C. Meister, Z. Su. “Structure-invariant testing for machine translation”. In: *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 2020: pp. 961-973.

[59] Y. Xiong, B. Wang, G. Fu, et al. “Learning to Synthesize”. In: *GI’18: Genetic Improvement Workshop*. 2018.

[60] J. Lei Ba, J. R. Kiros, G. E. Hinton. “Layer normalization”. In: *arXiv preprint arXiv:1607.06450* (2016).

[61] K. He, X. Zhang, S. Ren, et al. “Deep residual learning for image recognition”. In: *CVPR*. 2016: pp. 770-778.

[62] A. Vaswani, N. Shazeer, N. Parmar, et al. “Attention is all you need”. In: *NIPS*. 2017: pp. 6000-6010.

[63] M. Dehghani, S. Gouws, O. Vinyals, et al. “Universal transformers”. In: *arXiv preprint arXiv:1807.03819* (2018).

[64] F. Chollet. “Xception: Deep learning with depthwise separable convolutions”. In: *CVPR*. 2017: pp. 1251-1258.

[65] D. Hendrycks, K. Gimpel. “Bridging Nonlinearities and Stochastic Regularizers with Gaussian Error Linear Units”. In: *arXiv preprint arXiv:1606.08415* (2016).

[66] A. See, P. J. Liu, C. D. Manning. “Get To The Point: Summarization with Pointer-Generator Networks”. In: *ACL*. 2017: pp. 1073-1083.

[67] N. Shazeer, M. Stern. “Adafactor: Adaptive learning rates with sublinear memory cost”. In: *arXiv preprint arXiv:1804.04235* (2018).

[68] S. Lu, D. Guo, S. Ren, et al. “CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation”. In: *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*. 2021.

[69] L. S. Zettlemoyer, M. Collins. “Learning to map sentences to logical form: structured classification with probabilistic categorial grammars”. In: *UAI*. 2005: pp. 658-666.

[70] L. Zettlemoyer, M. Collins. “Online learning of relaxed CCG grammars for parsing to logical form”. In: *EMNLP-CoNLL*. 2007: pp. 678-687.

[71] T. Kwiatkowski, L. Zettlemoyer, S. Goldwater, et al. “Lexical generalization in CCG grammar induction for semantic parsing”. In: *EMNLP*. 2011: pp. 1512-1523.

[72] T. Kwiatkowski, E. Choi, Y. Artzi, et al. “Scaling semantic parsers with on-the-fly ontology matching”. In: *EMNLP*. 2013: pp. 1545-1556.

[73] A. Wang, T. Kwiatkowski, L. Zettlemoyer. “Morpho-syntactic lexical generalization for CCG semantic parsing”. In: *EMNLP*. 2014: pp. 1284-1295.

[74] P. Yin, G. Neubig. “TRANX: A Transition-based Neural Abstract Syntax Parser for Semantic Parsing and Code Generation”. In: *EMNLP*. 2018: pp. 7-12. URL: https://www.aclweb.org/anthology/D18-2002.

[75] B. Chen, L. Sun, X. Han. “Sequence-to-Action: End-to-End Semantic Graph Generation for Semantic Parsing”. In: *ACL*. 2018: pp. 766-777.

[76] K. Xu, L. Wu, Z. Wang, et al. “Exploiting Rich Syntactic Information for Semantic Parsing with Graph-to-Sequence Model”. In: *ACL*. 2018: pp. 918-924.

[77] Z. Sun, Q. Zhu, L. Mou, et al. “A grammar-based structural CNN decoder for code generation”. In: *AAAI*. Vol. 33. 2019: pp. 7055-7062. URL: https://doi.org/10.1609/aaai.v33i01.33017055.

[78] R. Murphy, B. Srinivasan, V. Rao, et al. “Relational pooling for graph representations”. In: *International Conference on Machine Learning*. 2019: pp. 4663-4673.

[79] R. Sato, M. Yamada, H. Kashima. “Random features strengthen graph neural networks”. In: *Proceedings of the 2021 SIAM International Conference on Data Mining*. 2021: pp. 333-341.

[80] D. P. Kingma, J. Ba. “Adam: A Method for Stochastic Optimization”. In: *ICLR*. 2015. URL: http://arxiv.org/abs/1412.6980.

[81] J. Pennington, R. Socher, C. Manning. “Glove: Global vectors for word representation”. In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 2014: pp. 1532-1543.

[82] Wikipedia. *Wikipedia*. https://dumps.wikimedia.org/. 2014.

[83] Robert Parker, David Graff, Junbo Kong, Ke Chen, Kazuaki Maeda. *English Gigaword Fifth Edition*. https://catalog.ldc.upenn.edu/LDC2011T07. 2011.

[84] SpaCy. *SpaCy*. https://spacy.io/. 2019.

[85] Ralph Weischedel, Martha Palmer, Mitchell Marcus, Eduard Hovy, Sameer Pradhan, Lance Ramshaw, Nianwen Xue, Ann Taylor, Jeff Kaufman, Michelle Franchini, Mohammed El-Bachouti, Robert Belvin, Ann Houston. *OntoNotes*. https://catalog.ldc.upenn.edu/LDC2013T19. 2013.

[86] C. D. Manning, M. Surdeanu, J. Bauer, et al. “The Stanford CoreNLP Natural Language Processing Toolkit”. In: *Association for Computational Linguistics (ACL) System Demonstrations*. 2014: pp. 55-60. URL: http://www.aclweb.org/anthology/P/P14/P14-5010.

[87] A. Taylor, M. Marcus, B. Santorini. “The Penn treebank: an overview”. In: *Treebanks*. Springer, 2003: pp. 5-22.

[88] J. Devlin, M. Chang, K. Lee, et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Ed. by J. Burstein, C. Doran, T. Solorio. Association for Computational Linguistics, 2019: pp. 4171-4186. URL: https://doi.org/10.18653/v1/n19-1423.

[89] X. Li, L. Bing, W. Zhang, et al. “Exploiting BERT for End-to-End Aspect-based Sentiment Analysis”. In: *Proceedings of the 5th Workshop on Noisy User-generated Text (W-NUT 2019)*. 2019: pp. 34-41.

[90] W. Zhou, T. Ge, K. Xu, et al. “BERT-based lexical substitution”. In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 2019: pp. 3368-3373.

[91] Y. Jia, M. Harman. “An Analysis and Survey of the Development of Mutation Testing”. In: *IEEE Transactions on Software Engineering* 37.5 (2011): pp. 649-678.

[92] M. Papadakis, M. Kintis, J. Zhang, et al. “Mutation testing advances: an analysis and survey”. In: *Advances in Computers*. Vol. 112. Elsevier, 2019: pp. 275-378.

[93] F. S. Foundation. *GNU Wdiff*. 2019. URL: https://www.gnu.org/software/wdiff/.

[94] J. W. Hunt, T. G. Szymanski. “A fast algorithm for computing longest common subsequences”. In: *Communications of the ACM* 20.5 (1977): pp. 350-353.

[95] E. S. Ristad, P. N. Yianilos. “Learning string-edit distance”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20.5 (1998): pp. 522-532.

[96] J. Gu, Y. Wang, K. Cho, et al. “Search engine guided neural machine translation”. In: *Thirty-Second AAAI Conference on Artificial Intelligence*. 2018.

[97] J. Zhang, M. Utiyama, E. Sumita, et al. “Guiding neural machine translation with retrieved translation pieces”. In: *arXiv preprint arXiv:1804.02559* (2018).

[98] Y. Zhang, R. Jin, Z H. Zhou. “Understanding bag-of-words model: a statistical framework”. In: *International Journal of Machine Learning and Cybernetics* 1.1 (2010): pp. 43-52. URL: https://doi.org/10.1007/s13042-010-0001-0.

[99] K. Papineni, S. Roukos, T. Ward, et al. “BLEU: a method for automatic evaluation of machine translation”. In: *Proceedings of the 40th annual meeting on association for computational linguistics*. 2002: pp. 311-318.

[100] S. Iyer, I. Konstas, A. Cheung, et al. “Mapping language to code in programmatic context”. In: *arXiv preprint arXiv:1808.09588* (2018).

[101] T. Wolf, L. Debut, V. Sanh, et al. “Transformers: State-of-the-Art Natural Language Processing”. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, 2020: pp. 38-45. URL: https://www.aclweb.org/anthology/2020.emnlp-demos.6.

[102] L. Backstrom, C. Dwork, J. M. Kleinberg. “Wherefore art thou r3579x?: anonymized social networks, hidden patterns, and structural steganography”. In: *WWW*. 2007: pp. 181-190. URL: https://doi.org/10.1145/1242572.1242598.

[103] D. Selsam, M. Lamm, B. Bünz, et al. “Learning a SAT Solver from Single-Bit Supervision”. In: *ICLR*. 2019. URL: https://openreview.net/forum?id=HJMC%5C_iA5tm.

[104] M. Allamanis, M. Brockschmidt, M. Khademi. “Learning to Represent Programs with Graphs”. In: *ICLR*. 2018. URL: https://openreview.net/forum?id=BJOFETxR-.

[105] J. Wei, M. Goyal, G. Durrett, et al. “LambdaNet: Probabilistic Type Inference using Graph Neural Networks”. In: *ICLR*. 2020. URL: https://openreview.net/forum?id=Hkx6hANtwH.

[106] Z. Chen, L. Li, J. Bruna. “Supervised community detection with line graph neural networks”. In: *International Conference on Learning Representations*. 2018.

[107] W. Azizian, et al. “Expressive power of invariant and equivariant graph neural networks”. In: *International Conference on Learning Representations*. 2020.

[108] Z. Wu, S. Pan, F. Chen, et al. “A Comprehensive Survey on Graph Neural Networks”. In: *IEEE Trans. Neural Networks Learn. Syst.* 32.1 (2021): pp. 4-24. URL: https://doi.org/10.1109/TNNLS.2020.2978386.

[109] R. Samdani, M W. Chang, D. Roth. “Unified expectation maximization”. In: *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2012: pp. 688-698.

[110] P. Boldi, F. Bonchi, A. Gionis, et al. “Injecting Uncertainty in Graphs for Identity Obfuscation”. In: *VLDB Endow.* (2012): pp. 1376-1387. URL: https://doi.org/10.14778/2350229.2350254.

[111] K. Hazelwood, S. Bird, D. Brooks, et al. “Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective”. In: *24th International Symposium on High-Performance Computer Architecture (HPCA 2018), February 24-28, Vienna, Austria*. 2018.

[112] J. M. Zhang, M. Harman, L. Ma, et al. “Machine Learning Testing: Survey, Landscapes and Horizons”. In: *arXiv preprint arXiv:1906.10742* (2019).

[113] Y. Belinkov, Y. Bisk. “Synthetic and natural noise both break neural machine translation”. In: *Proc. ICLR*. 2018.

[114] H. Khayrallah, P. Koehn. “On the impact of various types of noise on neural machine translation”. In: *arXiv preprint arXiv:1805.12282* (2018).

[115] V. Karpukhin, O. Levy, J. Eisenstein, et al. “Training on Synthetic Noise Improves Robustness to Natural Noise in Machine Translation”. In: *arXiv preprint arXiv:1902.01509* (2019).

[116] Parmy Olson. *The Algorithm That Helped Google Translate Become Sexist*. 2018.

[117] Y. Liu, M. Sun. “Contrastive unsupervised word alignment with non-local features”. In: *Twenty-Ninth AAAI Conference on Artificial Intelligence*. 2015.

[118] Google. *Google Translate*. http://translate.google.com. 2019.

[119] CWMT. *The CWMT Dataset*. http://nlp.nju.edu.cn/cwmt-wmt/. 2018.

[120] M. Ziemski, M. Junczys-Dowmunt, B. Pouliquen. “The united nations parallel corpus v1. 0”. In: *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*. 2016: pp. 3530-3534.

[121] WMT. *News-Commentary*. http://data.statmt.org/wmt18/translation-task/. 2018.

[122] A. Vaswani, S. Bengio, E. Brevdo, et al. “Tensor2Tensor for Neural Machine Translation”. In: *CoRR* abs/1803.07416 (2018). URL: http://arxiv.org/abs/1803.07416.

[123] H. Hassan, A. Aue, C. Chen, et al. “Achieving Human Parity on Automatic Chinese to English News Translation”. In: *CoRR* abs/1803.05567 (2018). arXiv: 1803.05567. URL: http://arxiv.org/abs/1803.05567.

[124] J. Hao, X. Wang, B. Yang, et al. “Modeling Recurrence for Transformer”. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, 2019: pp. 1198-1207. URL: https://www.aclweb.org/anthology/N19-1122.

[125] Y. Cheng, Z. Tu, F. Meng, et al. “Towards robust neural machine translation”. In: *arXiv preprint arXiv:1805.06130* (2018).

[126] M. T. Ribeiro, S. Singh, C. Guestrin. “Semantically equivalent adversarial rules for debugging nlp models”. In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2018: pp. 856-865.

[127] D. Pesu, Z. Q. Zhou, J. Zhen, et al. “A Monte Carlo Method for Metamorphic Testing of Machine Translation Services”. In: *3rd IEEE/ACM International Workshop on Metamorphic Testing, MET 2018, Gothenburg, Sweden, May 27, 2018*. ACM, 2018: pp. 38-45. URL: http://ieeexplore.ieee.org/document/8457612.

[128] J. Cao, M. Li, Y. Li, et al. “SemMT: A Semantic-based Testing Approach for Machine Translation Systems”. In: *CoRR* abs/2012.01815 (2020). arXiv: 2012.01815. URL: https://arxiv.org/abs/2012.01815.

[129] P. He, C. Meister, Z. Su. “Testing Machine Translation via Referential Transparency”. In: *arXiv*. 2021.

[130] S. Gupta, P. He, C. Meister, et al. “Machine translation testing via pathological invariance”. In: *ESEC/FSE ’20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. Ed. by P. Devanbu, M. B. Cohen, T. Zimmermann. ACM, 2020: pp. 863-875. URL: https://doi.org/10.1145/3368089.3409756.

[131] L. Sun, Z. Q. Zhou. “Metamorphic testing for machine translations: MT4MT”. In: *2018 25th Australasian Software Engineering Conference (ASWEC)*. 2018: pp. 96-100.

[132] P. He, C. Meister, Z. Su. “Testing Machine Translation via Referential Transparency”. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 2021: pp. 410-422.

个人简历及博士期间研究成果

个人简历

孙泽宇，1995 年 10 月出生于安徽省宣城市；2013 年 9 月考入北京化工大学信息科学与技术学院，专业为计算机科学与技术，2017 年 7 月本科毕业并获得工学学士学位；2017 年 9 月考入北京大学信息科学技术学院计算机软件与理论专业攻读硕士学位；2019 年通过硕转博攻读博士学位至今。

发表论文

[1] **Zeyu Sun**, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang, A Grammar-Based Structural CNN Decoder for Code Generation, The Thirty-Third AAAI Conference on Artificial Intelligence **(AAAI, CCF-A)**, 2019.

[2] **Zeyu Sun**, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang, TreeGen: A Tree- Based Transformer Architecture for Code Generation, The Thirty-Fourth AAAI Conference on Artificial Intelligence **(AAAI, CCF-A)**, 2020.

[3] **Zeyu Sun**, Jie M. Zhang, Mark Harman, Mike Papadakis, and Lu Zhang, Automatic Testing and Improvement of Machine Translation, The 42nd International Conference on Software Engineering **(ICSE, CCF-A)**, 2020.

[4] **Zeyu Sun**, Wenjie Zhang, Lili Mou, Qihao Zhu, Yingfei Xiong, Lu Zhang, Generalized Equivariance and Preferential Labeling for GNN Node Classification, Thirty-Sixth AAAI Conference on Artificial Intelligence **(AAAI, CCF-A)**, 2022

[5] **Zeyu Sun**, Jie M. Zhang, Yingfei Xiong, Mark Harman, Mike Papadakis, and Lu Zhang, Improving Machine Translation Systems via Isotopic Replacement, The 44th International Conference on Software Engineering **(ICSE, CCF-A)**, 2022

[6] Zheng Li, Bin Peng, Xiang Chen, **Zeyu Sun**, Yong Liu, Yonghao Wu, Deli Yu, SeCNN: A Semantic CNN Parser for Code Comment Generation, Journal of Systems and Software **(JSS, CCF-B, 通讯作者)**, 2021

[7] Wenjie Zhang, **Zeyu Sun**, Qihao Zhu, Ge Li, Shaowei Cai, Yingfei Xiong, and Lu Zhang, NLocalSAT: Boosting Local Search with Solution Prediction, the 29th International Joint Conference on Artificial Intelligence and the 17th Pacific Rim International Conference on Artificial Intelligence **(IJCAI-PRICAI, CCF-A)**, 2020.

[8] Xin Tan, Minghui Zhou, and **Zeyu Sun**, A First Look at Good First Issues on GitHub, The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering **(ESEC/FSE, CCF-A)**, 2020.

[9] Qihao Zhu, **Zeyu Sun**, Xiran Liang, Yingfei Xiong, and Lu Zhang, OCoR: An Overlapping-Aware Code Retriever, International Conference on Automated Software Engineering **(ASE, CCF-A)**, 2020

[10] 曹英魁，**孙泽宇**，邹艳珍，谢冰. 一种结构信息增强的代码修改自动转换方法. **(软件学报，CCF-中文 A)**，2021

[11] Qihao Zhu, **Zeyu Sun**, Yuan'an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, Lu Zhang, A Syntax-Guided Edit Decoder for Neural Program Repair, The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering **(ESEC/FSE, CCF-A)**, 2021.

[12] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, **Zeyu Sun**, Dan Hao, Lu Zhang, Lingming Zhang, Boosting Coverage-Based Fault Localization via Graph-Based Representation Learning, The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering **(ESEC/FSE, CCF-A)**, 2021.

[13] Jinhao Dong, Yiling Lou, Qihao Zhu, **Zeyu Sun**, Zhilin Li, Wenjie Zhang, Dan Hao, FIRA: Fine-Grained Graph-Based Code Change Representation for Automated Commit Message Generation, The 44th International Conference on Software Engineering **(ICSE, CCF-A)**, 2022

[14] 梁清源, 朱琪豪, **孙泽宇**, 张路, 张文杰, 熊英飞, 梁广泰, 郁莲. 基于深度学习的 SQL 生成研究综述. **(中国科学: 信息科学，CCF-中文 A)**，2022

[15] Qingyuan Liang, **Zeyu Sun**, Qihao Zhu, Wenjie Zhang, Lian Yu, Yingfei Xiong, and Lu Zhang, Lyra: A Benchmark for Turducken-Style Code Generation, the 31st International Joint Conference on Artificial Intelligence and the 25th European Conference on Artificial Intelligence **(IJCAI-ECAI, CCF-A)**, 2022.

[16] Qihao Zhu, **Zeyu Sun**, Wenjie Zhang, Yingfei Xiong, and Lu Zhang, Grape: Grammar Preserving Rule Embedding, the 31st International Joint Conference on Artificial Intelligence and the 25th European Conference on Artificial Intelligence **(IJCAI-ECAI, CCF-A)**, 2022.

参与课题

[1] 深度学习系统测试技术，国家重点研发计划政府间合作项目，No.2019YFE0198100

[2] 软件维护，国家自然科学基金优秀青年基金项目，No.61922003

致谢

燕园情，千千结，从 2017 年初入燕园，到如今的 2022 年，燕园陪伴我走过了五年的时光。感谢与燕园的相遇，在这五年里，我走过了燕园的每一个角落，喜欢着燕园的一草一木。而如今，我即将离开这生活了五年的园子，我会将在燕园的过往牢记于心，同时也感谢所有在燕园相遇的人们。

感谢杨芙清院士和梅宏院士。两位院士是北京大学乃至全中国的软件工程领域的开拓者，为中国的软件工程事业做出了巨大的贡献。我能够进入到两位院士领导的北京大学软件工程研究所攻读博士学位，是我莫大的幸运。是两位院士，让我可以有幸和众多优秀的学术前辈和同侪们合作，和各种喜爱学术的同学一起讨论最前沿的学术问题。

感谢我的导师张路教授。从我入学开始，张老师就一直引领着我的科研道路。让我从一个不懂科研的本科生，逐渐转变为拥有一些研究成果的博士。张老师对我在学术上的关怀无微不至，从学术问题的提出、分析、实验到撰写论文、亦或是学术报告等，张老师都给与了我很多帮助。永远都会记得张老师辛辛苦苦帮我修改论文的日子。他在学术上的严谨和博学，不断带领着我在学术道路上前进。在帮助我学术的同时，张老师的谦和、低调、真诚，也深深感染着我。我希望有一天我也可以成为张老师这样的好老师。

感谢我的协助指导老师熊英飞长聘副教授。熊老师则是我科研道路的塑造者。与熊老师刚认识的时候，他带我参观了实验室，使我感受到了实验室浓厚的科研氛围。在我入学之后，熊老师和张老师一起指导着我科研。在学术问题的研究上，熊老师经常和我讨论我遇到的难题，陪伴着我在学术道路上披荆斩棘。在学术观念上，熊老师曾经和我说过“我们要做改变世界的研究”，这种观念深深感染了我，不断激励着我朝着这个方向前进。在生活上，熊老师亦师亦友，时而也会和同学们一起游玩，陪伴着我们成长。能同时有着张老师和熊老师的指导是我莫大的荣幸。

感谢我所在小组里的老师郝丹教授。郝老师对待学术有着强烈的热情和严谨的态度。这样的热情和态度时常感染并激励着我前行。同时，郝老师也给予了我很多学术上的帮助，让我在科研道路上不断成长、进步。

感谢我的师兄阿尔伯塔大学助理教授的牟力立。牟师兄是我在科研道路上的长期合作者。从最早的帮我修改论文开始，牟师兄就指导着我如何撰写一篇英文论文。在后续无数次的远程讨论中，他也指导我如何做有影响力的研究，如何鉴别学术工作。他的指导和学术观点一直影响着我，帮助着我走过坎坷。

感谢我的师姐张洁。张师姐也是我科研道路上的长期合作者。在无数次的远程讨论中，张师姐对我科研给予了很多指导和帮助。在一起写作论文的时候，张师姐也指导着我如何撰写一篇好的英文论文，帮助着我提升我的写作水平。张师姐对科研的态度和见解也帮助着我完善我的科研道路。

感谢 Mark Harman 教授、Mike Papadakis 研究员，在与他们合作的过程中，我受到了他们的鼓励和指导。他们帮助我走过黑暗，并完成了一系列工作。

感谢金芝教授、周明辉教授、胡振江教授、谢涛教授、焦文品教授、王亚沙教授、王千祥教授、李戈副教授、邹艳珍副教授、陈泓婕副教授、张昕研究员以及我本科的李征教授、刘勇副教授。他们在我的博士培养过程中，给予了我许多宝贵的指导意见，让我不断成长。

感谢我所在小组里的王然、唐浩、陈俊洁、邹达明、娄一翎、冯致远、郭翊庆、Jianyi Zhou、王冠成、李丰、朱琪豪、张文杰、孙培艺、董谨豪、赵逸凡、孙可、张雅坤、吴宜谦、梁清源、肖元安同学。感谢其他小组里的陈震鹏、梁晶晶、陈潇漪、王敏、曹英魁、悦茹茹、涂菲菲、谭鑫、曾慕烙、申博、刘芳、刘炳言、林泽琦、杨恺、沈琦、姜佳君、王博、张星、陈蔚燕、王炫之、刘兆鹏、李洋、李晟洁、刘少钦、武健宇、高凯、王朝、王青叶、Yue Wang、张宇霞等同学。谢谢所有在学校的认识的同学们，他们在科研和生活上都给予了我帮助。同时也感谢桂媛媛同学帮我检查论文。

感谢一直在我身边的朋友们，他们陪我经历了人生中的喜怒哀乐。

感谢我的妈妈。我的妈妈在我的成长过程中给予了我无私的爱，一个人陪我从小学到我博士毕业。在我难过的时候，她一直鼓励我支持我前行。在我开心的时候，她和我一起分享快乐。她是我最坚强的后盾，也是我在黑暗中的亮光。

最后，感谢在燕园的五年时光。

北京大学学位论文原创性声明和使用授权说明

原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律结果由本人承担。

论文作者签名：孙泽宇　日期：2022 年 4 月 21 日

学位论文使用授权说明

（必须装订在提交学校图书馆的印刷本）

本人完全了解北京大学关于收集、保存、使用学位论文的规定，即：

- 按照学校要求提交学位论文的印刷本和电子版本；
- 学校有权保存学位论文的印刷本和电子版，并提供目录检索与阅览服务，在校园网上提供服务；
- 学校可以采用影印、缩印、数字化或其它复制手段保存论文；
- 因某种特殊原因需要延迟发布学位论文电子版，授权学校□一年/□两年/□三年以后，在校园网上全文发布。

（保密论文在解密后遵守此规定）

论文作者签名：孙泽宇　导师签名：张长水

日期：2022 年 4 月 21 日

学位论文答辩委员会名单

论文题目	感知关键约束的深度学习程序生成技术研究			
作　者	孙泽宇			
专　业	计算机软件与理论			
答辩委员会成员	**姓　名**	**专业技术职称**	**从事专业**	**工作单位**
主席	魏峻	研究员	计算机软件与理论	中科院软件所
委员	郝丹	教授	计算机软件与理论	北京大学
	刘辉	教授	计算机软件与理论	北京理工大学
	谢冰	教授	计算机软件与理论	北京大学
	熊英飞	研究员	计算机软件与理论	北京大学

北京大学博士学位论文答辩委员会决议书

（本表由博士答辩委员会秘书填写，一式两份，一份存学校档案，一份存研究生个人档案）

院、系：计算机学院　　专　　业：计算机软件与理论

姓　名：孙泽宇　　研究方向：软件工程与系统软件

学　号：1901111277　　导师姓名：张路

学位论文题目：　感知关键约束的深度学习程序生成技术研究

答辩委员会对学位论文和答辩情况的学术评语　（主要就论文选题意义，创新性成果及学术水平；论文存在的主要不足之处，以及博士生答辩情况等方面）。

程序生成是当前软件工程的研究热点。论文研究感知关键约束的深度学习程序生成技术，选题具有重要的学术意义和应用价值。

论文主要工作和创新成果包括：

（1）针对现有程序语法约束感知问题，提出了一种针对抽象语法树的树形Transformer结构，提升了对程序结构信息和标识符长程依赖的处理能力；

（2）针对深度学习处理标识符命名约束的问题，提出了一种感知标识符命名约束的迭代筛选算法，可通过优化训练数据提升深度学习模型的效果；

（3）针对任务特定的扰动约束问题，提出了一种深度学习代码生成中扰动约束的检测和修复算法，可通过后处理方式提升代码生成效果；

（4）技术推广到无属性图分类任务及机器翻译任务上，降低了预测错误率。

论文基于真实数据进行了实验分析，方案合理，结论可信。

论文写作规范、结构合理、叙述清晰、逻辑性强，是一篇优秀的博士论文。论文工作表明，孙泽宇同学已掌握了本领域坚实宽广的基础理论和系统深入的专门知识，独立从事科研工作的能力强。

孙泽宇同学在答辩过程中叙述清楚，回答问题正确。答辩委员会经过无记名投票，一致同意通过该同学的博士论文答辩，并建议授予博士学位。

答辩委员会表决结果：　实到答辩委员：　5　人，

同意博士毕业者　5　人，不同意博士毕业者　0　人，结论为　准予毕业

同意建议授予博士学位者　5　人，不同意建议授予博士学位者　0　人，结论为　建议授予学位

答辩时间：　2022 年 05 月 26 日 10 时

答辩委员会主席（签字）：

委员（签字）：

本表不得折叠、不得跨页

提交终版学位论文承诺书

本人郑重承诺，所呈交的学位论文为最终版学位论文。本人知晓，该版学位论文将用于校学位评定委员会审议学位、国家和北京市学位论文抽检。论文版本呈交错误带来的结果将由本人承担。

论文作者签名：孙泽宇

日期：2022 年 5 月 30 日