南京大学

本科毕业论文

院　　系　软件学院

专　　业　软件工程

题　　目　**一种基于注释分类的注释克隆检测方法**

年　　级　2019　学　　号　191820133

学生姓名　陆昱宽

导　　师　潘敏学　职　　称　副教授

提交日期　2023 年 6 月 1 日

南京大学本科生毕业论文（设计、作品）中文摘要

题目：一种基于注释分类的注释克隆检测方法
院系：软件学院
专业：软件工程
本科生姓名：陆昱宽
指导教师（姓名、职称）：潘敏学 副教授
摘要：

程序员在开发过程中经常会从其他项目中克隆代码并进行重构。理想情况是在修改代码的同时，程序员会同步地修改注释。然而开发人员有时会忽视相应注释的更新，导致注释与代码不一致。这种注释-代码的不一致性使得克隆代码块变得“脆弱”。注释在帮助程序员理解代码和不同部分之间的关系以及代码用法和交流方面起着重要作用。过时的注释可能会误导开发人员并引入错误，从而降低软件质量，因此是有害的。

本文专注于注释克隆检测技术的研究，探讨其在软件开发中的重要性与应用。注释克隆检测的目标是识别代码中的注释克隆，即具有相似或相同注释内容的代码片段。本文提出了注释克隆规则，并结合实现了注释克隆检测器，使用注释克隆检测技术来帮助检查代码-注释的不一致性。本文的注释克隆检测结合了注释分类技术和注释相似度检测技术，这是为了将代码的语义信息与注释相结合，使方法具备更好的可解释性。

具体而言，本文会将完整注释以语句为粒度拆分成子注释，首先对子注释进行注释分类，将子注释和代码的语义信息结合，然后对子注释进行克隆检测。子注释的克隆规则是本文提出的，并基于自然语言处理技术实现。

之后将子注释合并，属于同一注释段的子注释会被合并为完整注释。完整注释的样本会具有子注释的分类和克隆检测信息。通过上述处理，本文从注释的自然语言形式中提取出了子注释分类和子注释相似度信息，这些信息涉及注释的语义，结构以及代码的语义。

接下来，本文提出注释克隆检测的原则，并应用这些原则对一个大规模的数据集进行人工标注。注释克隆检测器采用神经网络进行实现。处理后的数据集会被用于训练神经网络，训练后的模型被作为注释克隆检测器。基于上述工作，本文还实现了原型工具，并设计了实验方法，通过实验来评估工具的有效性。

关键词：注释克隆；代码-注释一致性；注释分类

南京大学本科生毕业论文（设计、作品）英文摘要

THESIS: Comment clone detection technology and its application
DEPARTMENT: Software Institute
SPECIALIZATION: Software Engineering
UNDERGRADUATE: Yukuan Lu
MENTOR: Associate Professor Minxue Pan
ABSTRACT:

Programmers often clone code from other projects and refactor it during the development process. Ideally, as the code is modified, the programmer will change the comments in parallel. However, developers sometimes neglect to update the comments accordingly, resulting in inconsistencies between the comments and the code. This comment-code inconsistency makes cloned blocks of code 'fragile'. Comments play an important role in helping programmers understand the code and the relationships between different parts, as well as code usage and communication. Outdated comments can be harmful as they can mislead developers and introduce errors that can degrade software quality.

This thesis focuses on the study of comment clone detection techniques and explores their importance and application in software development. The goal of comment clone detection is to identify comment clones in code, i.e. code fragments with similar or identical comment content. In this thesis, comment clone rules are proposed and combined with the implementation of a comment clone detector to help check for code-comment inconsistencies. This thesis combines comment classification techniques with comment similarity detection techniques in order to combine the semantic information of the code with the comments and to make the approach more interpretable.

Specifically, the complete comment is split into sub-comments at the granularity of statements, the sub-comments are classified, the semantic information of the code is combined with the sub-comments, and then the sub-comments are cloned. The cloning rules for sub-comments are proposed in this thesis and are based on natural language processing techniques.

Sub-comments are then merged, and sub-comments belonging to the same comment segment are merged into full comments. The samples of the full comment will have the classification and cloning detection information of the sub-comments. Through the above process, we extract the sub-comment classification and sub-comment similarity information from the natural language form of the comment, which relates to the semantics, structure and code semantics of the comment.

The thesis then proposes principles for comment clone detection and applies these principles to a large-scale dataset for manual comment. The comment clone detector is

implemented using a neural network. The processed dataset is used to train the neural network, and the trained model is used as the comment clone detector. Based on this work, a prototype tool is implemented and an experimental approach is designed to evaluate the effectiveness of the tool through experiments.

Keywords: Comment clone; code-comment consistency; comment classification

目　录

插图清单

表格清单

第一章　绪论

1.1　研究背景

注释在软件开发过程中扮演着至关重要的角色，它们提供了对代码功能和设计意图的解释，有助于开发人员理解和维护代码。当代码发生克隆时，注释也会随之被克隆。然而，当开发者修改代码时，很可能忽视修改相应的注释，从而引入潜在的错误[1-3]。现代软件系统通常包含大量的代码注释[4]，注释代码已经被认为是一种良好的编程实践[1][2]，它既有利于代码的编译，也有利于软件维护。例如，Tenny 等人[5-6]进行的实验表明，代码注释可以帮助提高代码的可读性，而 Jiang 等人[7]则证明，代码注释在维护软件方面起着重要作用。此外，代码注释提供了丰富的内涵，可以用来帮助执行广泛的软件工程任务，如错误检测[8-10]、规范推理[11-13]、测试[14-15]和代码协同。

对这种克隆的注释的检测存在着许多困难，因为注释是用自然语言书写的，没有固定的结构，其内容也没有固定的约束。注释可以描述代码的不同方面，例如方法的实现步骤，提供方法的原因，使用方法的注释事项等等。注释也可以对代码的不同部分进行描述，例如描述方法的签名，返回值，参数等等。

因此，传统的基于规则的模型并不能很好地处理注释更改的所有情况。对于基于机器学习的模型，数据质量一直是一个重大的问题。Xu 等人[16]指出，在基于机器学习的过时注释处理中，数据质量会对模型的结果有明显影响，同时由于现实世界中存在着大量的大型代码库，对这些代码库进行数据提取，构建出数据集也需要花费很多时间。基于机器学习的模型能够处理大规模的代码-注释变更情况，由于没有人工制定的规则的限制，因此也能够对各种类型的注释予以处理。但是，也正因为缺少人工制定的规则，基于机器学习的方法并没有良好的可解释性。他们对数据样本的标注的准确性往往需要人工实验来进行验证。

注释分类指的是按照注释的文本特征，语义内容，内在结构以及与代码语义的关联对注释建立分类法并进行分类，在这方面已经有了一些工作，Padioleau 等人[4]基于注释的语义提出了一个注释分类法。Steidl 等人[1]研究了注释分类，以便为注释质量评估提供更好的量化见解。为机器学习技术自动分类注释而手动给出特征。在[17-18]中的研究人员首先手动分类了 2000 多条代码注释，然后使用监督学习来达到大约 85% 的分类准确率。然而，他们的分类法没有结合程序分析，因此无法很好地被用于处理代码-注释的一致性，因为这个问题同时也涉

[1]http://universaldependencies.org/docsv1/u/dep/ case.html

[2]https://issues.apache.org/jira/ browse/COLLECTIONS- 727

及到代码的更改。Zhai 等人[19]提出了一个复杂的分类法，将注释分析和程序分析相结合，并且使用机器学习技术，可以处理现实的大型代码库中的复杂情况。

本文提出了注释克隆检测技术来处理这一问题，维护软件系统中的注释质量。本文提出了注释克隆的定义，并进行了注释克隆检测工具的实现。目前暂时没有注释克隆的先前工作，但是，在代码克隆检测领域已经又了很多工作，而注释克隆检测和代码克隆检测技术有一定的相似性。

代码克隆检测是指在代码库中发现相同或相似的代码片段。基于抽象语法树和依赖图的代码检测工具可以利用代码中的语义信息进行检测，但由于性能原因，时间复杂度较高，无法应用于大型代码库。传统的基于标记的检测技术可以对源代码进行快速的顺序检测，但是缺乏足够的语义信息，因此对于复杂的克隆情况难以有较好的效果。

1.2 研究现状及动机

现代软件系统开发中，克隆代码并进行本地修改的现象频繁地发生。当程序员在修改代码后忘记修改相应的注释，便发生了注释克隆情况，导致了代码-注释一致性的破坏。这种不一致性在软件系统中很普遍，有时甚至会造成严重的后果。本文用一个案例来说明注释克隆情况，例子如下:

```
1  /**
2   * @param loadedClass: 泛型，被加载类
3   * @return 布尔值，表明该类是否为抽象类
4   */
5  private static boolean isAbstractModule(Class<?>
6      loadedClass)
7  {
8          ...
9  }
```

上例中，本文给出了 *isAbstractModule*（*Class cls*）i 方法，该方法用于测试参数（类型）是否是一个抽象类。当开发者克隆这份代码到自己的项目后，可能会:

1. 修改方法，加入额外的逻辑，比如，如果参数是抽象类，则向标准输出输出一段内容。
2. 修改注释，如再添加或删除一段注释，比如添加说明该方法内部逻辑的注释。或对已有的注释内容做修改。
3. 同时修改代码和注释，比如新增加一个方法参数，并同步更新注释。

Fluri 等人 [27]，[28] 发现，由代码变化引发的注释变化中有超过 97% 是在该代码变化的同一次更改中完成的，换句话说，开发者有 97% 的概率在修改代码后同步更新注释，仅有 3% 的概率单独修改注释，不修改代码。因此（2）的情况是比较少见的。

对于（3），本文认为当开发者更新代码的同时更新注释时，新注释和新代码是一致的，除非开发者出于自己的疏忽写错了新注释，后者属于注释和代码的语义性一致，不属于本文的讨论范畴。

总而言之，本文只考虑开发者单独修改代码并忘记更改注释的情况，即情况（1）。本文中将这种情况看作注释克隆，它导致了代码-注释一致性的破坏，并且是许多软件问题的原因。

1.3 本文主要工作

本文提出了注释克隆检测方法，从注释的角度对代码-注释一致性进行检查。本文提出了一系列的注释克隆规则，这种基于规则的方法的优点是使方法能具有一定的可解释性。本文通过人工标注的方式应用这些规则，并且使用神经网络技术，基于标注好的大规模数据集进行模型训练，使用训练好的模型进行自动化的检测。此外，为了在注释的处理部分能够兼顾代码的语义，本文使用了Zhai[19]等人提出的注释分类方法 CPC 对注释进行预先的分类。为了让工具具备良好的泛化能力，本文使用一个大规模的数据集对神经网络模型进行训练。最后，为了提升数据质量，降低噪声样本对方法实现的影响，本文还进行了一些数据集清洗工作。总的来说，本文的工作主要有以下几个部分:

1. 提出了注释克隆检测的思想，并提供了具体的定义。注释克隆检测是基于注释分类和子注释克隆检测实现的。
2. 参考代码克隆的定义，提出了子注释克隆的定义，将子注释克隆分为四个级别，并对每个级别的检测分别予以实现。
3. 实现了方法的原型工具，并通过实验验证了方法的效果。

此外，虽然本文将注释克隆检测用于代码-注释一致性的研究，但是注释克隆检测本身是一个独立的领域，可以用在许多其他用途。例如代码查重，代码版权信息检查和软件系统安全性评估。

1.4 本文组织结构

本文的内容按如下结构组织:

第一章，绪论。主要阐述本文的研究背景，注释克隆的基本思想，研究动机以及相关方向的研究工作。

第二章介绍注释克隆和代码-注释一致性检测技术的背景知识和相关工作，并说明了注释克隆技术使用和参考的主要思想，包括注释分类，过时注释检测和代码克隆。注释克隆检测指的是在给定代码-注释更改的情况下，判断新注释（对应更改后的代码）和旧注释（对应更改前的代码）是否具有克隆关系。如果一个代码块拥有具有克隆关系的注释，该代码段被认为是容易存在代码-注释不

一致的，即容易存在缺陷的。本文对注释克隆的检测分为语句粒度的注释克隆检测和完整注释克隆检测两个步骤。

第三章介绍语句粒度注释的克隆检测技术。本文的注释克隆检测方法分为两步骤：按语句粒度切割的子注释和完整注释。在这一章，阐述对子注释的处理方法，详细介绍了在这个过程中所采用的注释分类和子注释克隆检测技术。

第四章介绍了完整注释的克隆检测思想和技术，并说明了如何将注释克隆检测技术应用于现实中的大型软件系统。本章阐述了完整注释克隆检测的基本原则，以及整个注释克隆技术应用的方法框架和实现细节。实现细节包括数据标注的方法，数据集的选取和过滤方法，神经网络的基本组件和训练原理。

第五章，工具实现与实例研究。首先介绍了本文实现的原型工具的架构，以及对该工具的评估。并且使用了多个实验度量来回答对注释克隆检测工具的研究问题。此外还对实验环境做出了说明，方便实验的复现。

第六章，总结与展望。本章是对全文工作的总结，并且对未来可能要进行的扩展工作进行了展望。

第二章　相关研究工作

在本节中将回顾与注释克隆和代码-注释一致性检测相关的先前研究工作。这些研究致力于解决克隆代码块中的代码-注释不一致性问题，并提出了各种方法和技术。

2.1　代码-注释一致性检测面临的挑战

之前的研究从不同的角度调查了源代码和代码注释之间的一致性变化[2-3,7,20-22]。例如，Jiang 和 Hassanc[7]发现 PostgreSQL 中注释函数的比例随着时间的推移保持稳定。Fluri 等人[21-22]强调，API 注释经常被追溯性地调整。Ibrahim 等人[20]调查了三个开源系统的注释更新和软件错误之间的关系，发现异常的注释更新行为是预测软件错误错误的一个很好的指标。Linares-Vasquez 等人[2]研究了开发者如何在方法注释中记录数据库使用情况，并指出数据库相关方法的注释比源代码的更新频率要低。Wen 等人[3]进行了一项大规模的实证研究，分析了不同的代码变更类型触发注释更新的概率。他们还提出了一个代码-注释不一致的分类法。这些工作可以分为两类：

1. 分析代码-注释不一致情况出现的原因（Fluri 等人，2009；Jiang 和 Hassan，2006；Ibrahim 等人，2012；Fluri、Wursch 和 Gall，2007）。
2. 分析不同类型的代码-注释不一致（Wen 等人，2019）。

如何分析代码-注释的一致性是一个复杂的问题，目前的研究方法主要着眼于代码克隆检测以及基于规则的或者基于机器学习的过时注释检测和更新。然而，这些方法分别存在一些问题。

对于代码克隆检测，它只关注代码块中的代码部分，而忽略了相应的注释部分。在下文本文会看到，相当多的注释-代码不一致是由于修改代码后注释没有得到相应更新而导致的，也就是说问题主要存在于注释部分，而非代码部分，代码克隆检测无法有效地解决这种情况。

过时注释的检测和更新主要有两种方法: 基于规则的模型采用数据流分析，通过人为定义的规则来捕捉代码中的一些特征，并且与注释中的特征相结合，检测并更新过时的注释。这样做的好处是能够将注释和代码有效地结合，将二者的特征信息结合起来。代码特征一般是在代码编译的中间层面捕获的，例如代码的标记（Token），抽象语法树（AST）或者中间代码（IR）。但是，由于规则是人为定义的，而过时注释的检测问题在现实可能变得相当复杂，因此基于规

则的模型并不能有效地处理过时注释检测的所有情况。另一种方法是采用基于机器学习的模型，这些模型采用复杂的神经网络，通过使用特定的嵌入层，将代码，代码修改实例，注释，注释修改实例等信息转化为低维的词嵌入，进而输入到神经网络中进行训练。需要注意的是，为了将注释和代码联系起来，有些方法会在嵌入层将注释和代码合成为统一的词嵌入，也有方法通过在神经网络中加入共同注意力层（co-attention layer）来使网络学习到这二者的信息。基于机器学习的方法的表达能力更强，也不会地受到既定规则的制约，因此能达到和基于规则的方法一样甚至更好的效果，这是由实验证实的。但是，基于机器学习的方法有两个重大缺陷。首先是模型的效果好坏受到所使用的数据集的质量的影响很大，低质量的，含有错误标记样本的数据集会显著降低模型的分类能力。其次是基于机器学习的方法相比基于规则的方法，没有良好的可解释性。

2.1.1 基于规则和基于机器学习的方法的对比

目前关于代码-注释一致性检测的研究主要分为基于机器学习和基于规则两个方向。Fraco[23]是一个基于规则的方法，用于检测由标识符重命名引起的代码-注释不一致。由 Liu 等人[24]提出的 CUP2，Xu 等人[16]提出的 ADVOC 都是基于深度学习的方法，它们将方法级别的代码-注释变更作为数据集。CUP2 和 ADVOC 都取得了优于基于规则的方法的效果，Liu 等人[24]认为，基于机器学习的方法有两大优点:

1. 得益于神经网络模型的强大表示能力，基于机器学习的方法可以从大规模的代码-注释更改样本中学习，而基于规则的方法只能处理特定类型的代码-注释更改。例如，Fraco 只能处理与标识符重命名有关的代码更改。实验也证明，无论是 CUP2 还是 ADVOC，都具有比基于规则的工具更好的表现能力，它们的覆盖率和准确率都更高。
2. 基于机器学习的方法可以在神经网络中加入特殊的组件，来捕获代码更改和注释之间的关系，例如词汇引用和语义相关性。

与此同时，Xu 等人[16]指出，基于机器学习的方法是数据驱动的，除了所采用的模型之外，方法所采用的数据集的质量对方法效果也有影响。对于代码-注释一致性检测这一任务来说，错误的数据样本可能使得模型对应该检测出的不一致样本感到不那么敏感，也会[9]使得模型的有效性无法被准确评估。

2.1.2 注释和代码语义的结合

要检测代码和注释的一致性，就需要考虑代码变化和注释变化之间的联系，在注处理注释的同时处理代码的语义信息。对于基于机器学习的方法，由于代码和注释是不同的文本结构，因此需要将二者结合，转化为统一的结构表示，才

能输入模型。之前的工作采用层次化的编码器，将数据样本的代码和注释部分进行差异化的编码，并且使用注意机制来提取二者之间的联系。

以 CUP2 为例，这是一个基于神经网络的代码-注释一致性检测和更新方法。它的数据样本是从多个大型现实中存在的大型代码库中提取而来的代码-注释对更改样本，每个样本包含: 旧代码片段，对应的新代码片段，与旧代码关联的旧注释和对应的新注释。CUP2 使用了一个 4 层的网络结构，分别是: 嵌入层，上下文嵌入层，共同注意层和建模层。

第一层是嵌入层，使用 fasttext 工具将注释和代码分别转化为词嵌入。第二层是上下文嵌入层，使用双向 LSTM 编码器提取出代码更改的词嵌入和注释更改的上下文向量，在这一步，代码和注释的更改依然是独立表示的。第三层是共同关注层，CUP2 将前一层输出的代表代码更改和注释更改的词嵌入结合起来，使用点积注意机制计算出特征向量并输出。该特征向量就代表了代码-注释的编辑，并且突出了注释更改中与代码更改有关的信息。在第四层通过双向 LSTM 编码器，根据第二层输出的上下文向量和第三层输出的编辑特征向量计算得到得到代码-注释更改的最终表示。

可以看到，CUP2 将代码和注释分别处理并转化为词嵌入，然后使用共同注意网络将代码和注释的信息结合起来，以得到代码-注释变化的统一表示。

ADVOC 采用了另一种编码方式，他们采用和 CUP2 相似的数据样本，但是使用了不同的词嵌入方式。具体而言，ADVOC 会对代码部分采取三个不同的词嵌入方式，分别是: 编辑序列，编辑树和编辑修改。他们认为不同的词嵌入形式可以帮助神经网络从不同的 “角度” 理解代码变化，例如，编辑序列是为了代码的自然性，编辑树揭示了更多的语法，而编辑脚本则直接提供了哪些实体被改变。下面将详细描述每种表示方法。

ADVOC 之后会对三种代码的编辑表示以及注释进行单独地编码，并且使用注意力机制来将它们转变为统一的词嵌入来代表数据样本。

更早的研究 Just-In-Time[25]也对多种形式的代码变化进行编码。它与 ADVOC 的关键区别有两个方面。首先，ADVOC 对旧注释、编辑序列、编辑树和编辑脚本进行建模，而 Just-In-Time 只对前三种进行建模。其次，ADVOC 还使用 Bi-GRU 来融合编辑树上每个节点的属性信息，并使用图注意来明确地编码旧注释和编辑树之间的联系。

2.1.3 数据集质量问题

Xu 等人[16]提出一种数据清洗方法，来提高基于机器学习的方法的数据集质量，提高模型的效果。他们首先检查数据集中的样本，并定义一系列数据清洗的规则，通过应用这些规则，达到缩减数据集并提高数据集准确性的目的。他们采用了一个典型的数据清洗流程，即发现错误，自动化检测错误和修复错误，该方法可以关注样本的实际语义，而不仅仅在语法层面进行修正。他们的实验

证明了数据清理确实可以提升模型的有效性，并且他们的方法可以有效地清除假阳性的样本。

2.2 过时注释检测

研究人员已经对过时注释检测进行了广泛的研究。其中，大多数研究集中在与特定代码属性或特定类型相关的注释[26]。Tan 等人[9-10]提出了一系列方法，利用手动定义的规则、NLP 技术和静态程序分析，检测与特定编程概念相关的代码注释不一致，例如锁机制[9,27]函数调用[9]和中断[28]。Tan 等人还设计了 @TCOMMENT 方法[10]，利用启发式方法和自动测试生成来检查 Java 方法和其 Javadocs 在方法参数对空值的容忍度方面的不一致。Sridhara 等人[29]提出了一种基于信息检索、语言学和语义学的技术，用于识别过时的 TODO 注释。

此外，还有一些研究针对一般的注释，并考虑了代码的变化[23,30]。例如，Ratol 和 Robillard[23]提出了一种名为 Fraco 的基于规则的方法，用于检测标识符重命名方面的脆弱注释。Malik 等人 [32] 经验性地研究了更新修改后的函数注释的合理性，考虑了修改后的函数的特征、修改本身、时间以及代码所有权。Liu 等人[24]利用 Random Forest 算法和 64 个人工制定的特征，从代码、注释和代码-注释关系中检查当一个块/行注释的相关代码片段被改变时是否要更新。Malik 等人[30]使用随机森林分类器预测注释是否会被更新，该分类器利用表面特征来捕捉被改变的方法、改变本身和所有权等方面。他们不考虑现有的注释，因为他们的重点不是不一致的检测；相反，他们旨在通过分析有用的特征来了解注释更新做法背后的理由。Liu 等人[31]提出的随机森林分类器是一个基于规则的方法，用于对方法级别的行粒度和块粒度的过时注释进行分析。Liu 等人[24]提出的过时注释检测技术（OCD）是一个端到端的解决方案，OCD 属于他们提出的两阶段过时注释检测-更新方法 CUP2 的一部分。与传统方法相比，OCD 有三个方面的区别。首先，OCD 利用了完整的代码上下文信息，使用代码-注释变更作为数据样本，而传统方法采用某一版本的代码作为数据样本，其次，OCD 可以处理任意类型的注释和代码变化。最后，OCD 是一个基于机器学习的工具，可以自动从大量的数据样本中学习。相比基于规则的工具，基于机器学习的 OCD 不需要遵守人工制定的规则，因此拥有更强的表达能力，能够处理更加多变的代码-注释更改情况。OCD 采用双向长短时记忆网络（Bi-LSTM）作为训练层进行模型训练，这是一种递归神经网络（RNN）的变体，专门用于处理序列数据，在自然语言处理领域有广泛的应用。双向长短时记忆网络由两个隐藏层组成，分别是前向隐藏层和后向隐藏层。每个隐藏层都是一个独立的 LSTM 结构，用于分别处理正向和逆向的序列数据。它可以同时处理前向和后向的上下文信息。正向隐藏层从序列的开头开始处理，而逆向隐藏层从序列的结尾开始处理，相比单项长短时记忆网络可以捕捉到序列中更全面的上下文信息，更好地理解序列数

据中的依赖关系和语境。

Xu 等人[16]提出的 ADVOC 在 OCD 的基础上做出了改进。ADVOC 使用了 OCD 的数据集，，并使用自己的数据清洗方法优化该数据集，ADVOC 采用多个编码层嵌入方式将代码-注释变更转化为词嵌入，并使用生成对抗网络训练模型，取得了优于 OCD 的效果。与传统的生成对抗网络不同，ADVOC 采用一个分类器来代替生成器，为每个不可靠的样本预测一个更合适的标签。换句话说，整个生成对抗网络由两个更小的神经网络组成，即一个分类器和一个鉴别器，相互对抗，以更好地揭示不可靠的样本集的标签信息。

2.3 注释分类

代码注释是软件开发者的一个重要信息来源。代码注释记录了功能，描述了用法，或者是开发者留下开发笔记的地方，这有助于维护性 [1] 和持续的开发任务 [2]。代码注释在帮助开发者理解源代码方面也发挥了很大作用 [1]。

现代软件提供了丰富的自然语言（NL）注释，并且在分析 NL 注释以及在软件工程中应用 NL 注释这两个方面已有大量的现有工作。然而现有的工作几乎没有利用程序分析技术来系统地导出、整理和传播注释。这种传播的注释包含丰富的语义，超出了广泛用于程序分析（如数据类型）的传统工件。例如，通过使用程序分析技术可以将注释传递给未注释的代码实体，并利用传播的注释中包含的信息来检测代码错误。

由于缺乏编写文档的标准，开发人员有很多的词汇可以选择，并且他们倾向于使用任意的方式来编写文档。他们通常对代码元素的不同方面进行注释，如类、方法和变量，并使用注释来描述各种内容，如总结功能、解释设计原理和指定实现细节。此外，由于注释是用自然语言写成的，它们本质上是模糊的，需要准确的语言分析来获得它们的确切含义和范围。为了更好地理解代码和更有效地传播注释，必须首先知道他们注释的是哪些代码元素，以及他们传达的是什么样的信息。也就是说需要要设计一个细致入微的代码注释分类法，并开发一个可靠的分类器来自动对注释进行分类。

在软件文档分类方面已经有了一些努力。Padioleau 等人[4]建立了一个基于注释含义的分类法。Maalej[32]的工作提出了 API 参考文献中知识类型的分类法，并使用该分类法来评估其包含的知识。在这个分类法的基础上，研究人员[33]开发了一套文本特征来自动分类知识。Steidl 等[1]研究了注释分类，以便为注释质量评估提供更好的量化的见解。Pascarella 等人[17-18]首先手动分类了 2000 多条代码注释，然后使用监督学习达到了大约 85% 的分类准确率。他们的分类法没有被设计成与程序分析相结合。目前还不清楚如何根据他们的分类来传播和推断注释。Padioleau[4]提出了一种基于以下四个维度的分类法：注释内容、注释的作者、注释的位置和注释撰写时间。Robillard 等人[32]根据知识类型（例如，功

能、概念、指令和代码示例）手动对 API 文档进行分类，以帮助人们理解和衡量 API 文档的质量。他们还研究了不同类型注释的分布。Pascarella[17]和 Steidl[1]的分类法相似，都包括目的（代码的功能）、开发中（与正在进行/未来开发相关的主题）和元数据（作者、许可证等）等类别。生成它们是为了促进注释的质量分析。Haouari[34]的分类法被提议用于调查开发人员的文档模式，而 Steidl[1]研究注释分类以提供关于注释质量评估文档的更好的定量见解。然而，它们的分类法不区分不同代码实体的注释，也不设计用于与程序分析相结合。目前尚不清楚在哪里以及如何根据注释的分类传播和推断注释。由 Al-Kaswan 等人提出的 STACC[35]是一个高性能的基于机器学习方法的注释分类器，能够对包括 Java，Python，Pharo 在内的多种语言进行注释分类，它通过使用 Setfit 的 Optuna 后端来进行超参数的自动优化，以提高模型性能。STACC 在较小的训练数据下就能够达到很高的准确性。Zhai 等人[19]提出的 CPC 从不同的角度（如代码的性质和代码的功能）和不同的代码实体（如类和方法）建立了一个注释分类法，并应用在多个开源代码库。CPC 的注释分类是由研究者手动观察并总结得到的。研究者先启发式地制定几条注释分类原则，然后多个实验人员手动检查代码库并且对注释分类，并且将各自的结果相互对比，研究者会不断地总结和迭代注释分类原则，直到现有的注释分类原则使所有研究者满意，并且不会在之后的分类中出现新的分类。

2.4 代码克隆

代码片段通常是源代码的一个连续段，它可以包含一个函数、代码块块或一连串的语句[36]。克隆对是一对代码片段[36]），它们在语法上或语义上彼此相似。

现有的研究一般把代码克隆分为四种类型[36-37]：

1. 第一类（文本相似性）：语法上完全相同的代码片段，除了空白处，注释和布局上的差异。
2. 第二类（词汇相似性）：除了 Type-1 克隆的差异，语法上相同的代码片段，除了标识符名称和字面价值的差异。Type-2 克隆反映了词法标记的差异。
3. （语法相似性）：也被称为近似克隆或间隙克隆。除了 Type-1 和 Type-2 克隆的差异外，句法上相似的代码片段在语句层面上也有差异。
4. 第四类（语义相似性）：除了 Type-1，Type-2 和 Type-3 克隆的差异外，代码片段可能在语法结构上不相似，但执行的是相同的任务。

在上面四个类型的划分的基础上，Sajnani 等人[37]进一步将第三类和第四类克隆分为四个细粒度的类型: 非常强类型（VST3）的相似性范围在 [0.9，1.0]，强类型-3（ST3）的相似性范围在 [0.7，0.9]，中等类型-3（MT3）的相似性范围在 [0.5, 0.7]，以及弱类型-3, 类型-4（WT3/T4）的相似性范围在 [0.0, 0.5]。

代码克隆检测对于代码的可维护性[38]、抄袭检测[39]，错误检测[40]恶意软件检测[36]都有很大作用，现有的研究报告表明，代码克隆在大型系统中很常见，Kamiya 等人[41]报告了 JDK 中 29% 的克隆代码。

代码克隆检测有许多难点。首先是某些代码的更改并不会影响程序的功能，代码克隆检测器需要对这样的情况不发生误判，例如 if 结构和 else 结构的位置可以互相调换，这样的程序的数据依赖性没有改变，因此没有改变程序的功能特性。再比如软件开发者通常会复制一个代码片段，并在行上做一些修改，或增加一些新的行来实现新的功能。代码克隆检测器需要对这种细粒度的代码更改做出判断。

已经有了许多代码克隆检测方面的研究，按使用的工具可以分为四类:

1. 传统的基于标记的克隆检测工具。
2. 基于 AST（Abstract Syntax Tree）的工具[42]。
3. 基于 PDG（Program Dependency Graph）的工具[43]可以检测接近错误的克隆（即、语法和语义信息的克隆（即有小的差异或差距的克隆（Roy 和 Cordy，2007）），但在处理子图同构等问题时耗时较大，这使得它们难以应用于大型代码库。
4. 基于深度学习的工具[44]。

基于标记的方法运行速度较快，资源占用更少，因此可以扩展到大型代码库。许多基于标记的工具使用一系列的策略来快速查询候选代码块。SourcerCC[37]利用低频标记上的优化倒置索引，设计过滤启发式方法来快速检测克隆。CCAligner（Wang 等人，2018）和 LVMapper[45]通过在代码行上滑动建立 k-line 索引，并使用该索引定位候选块，这将大大减少克隆比较的数量。但是这种方法也存在着明显的缺陷。由于将代码转化为标记序列后会丢失代码的语法信息和语义信息，这种工具只能检查较少的克隆情况。基于标记的方法可以根据检测粒度分为两类，包括词粒度（如 SourcerCC[37]）和行粒度（如 CCAligner[46]），NiCad[47]）。字粒度的方法将一个标记作为检测克隆的基本单位。SourcerCC 将源代码转换为标记序列。然后，它通过相同令牌的比率来验证克隆。行的粒度工具将一个规范化的行作为检测克隆的基本单位。CCAligner 通过在行上滑动代码窗口，设计了一个新的 e-mismatch 指数来检测大间隙的克隆。行粒度的方法将一个规范化的行作为检测克隆的基本单位。CCAligner 通过在行上滑动代码窗口，设计了一个新的 e-mismatch 指数来检测大间隙的克隆。NiCad 计算规范化源代码上最长的公共序列的长度来获得克隆，这使得它不能很好地检测混淆的克隆。基于语法树和基于依赖图的方法拥有比基于标记的方法更强的表达能力，因此可以处理更多的情况，但相对的，它们也使用了更加复杂的数据结构，导致这些方法的运行速度较慢，占用资源更多，对大型代码库分析的扩展性不好。语法树携带的程序上下文信息比标记多，而依赖图携带的程序上下文信息又比语法树多。因此三者的表达能力由小到大分别是: 标记，语法树，依赖图。而三者的运行效

果则依次递减。基于机器学习的工具通过数据集来进行模型训练，如前所述，这种方法的效果受数据集的影响很大。不仅需要数据集的规模足够庞大，还需要该数据集有良好的标签质量。构造数据集需要对大型代码库进行特征提取，这会消耗大量的时间。

2.5 本章小结

本章主要介绍两方面的内容:

1. 本文阐述了代码-注释一致性研究的现状以及它面临的挑战。这些挑战主要包括了基于规则和基于机器学习的方法的选择，如何有效地结合代码和注释的信息，以及如何提高数据集的质量。本文介绍了现有的对这些问题的解决方案。
2. 本文介绍了与本文研究相关的工作，主要包括了: 过时注释检测，注释分类和代码克隆。本文提出了注释克隆检测方法，并且将其与注释分类方法结合以获得更好的效果，注释克隆分类方法受到了代码克隆分类的启发。过时注释检测是代码-注释一致性检测的一部分，本文通过提出的注释克隆检测方法来解决该问题。

第三章　语句粒度注释的克隆检测

本文提出的注释克隆检测技术将完整注释按语句为粒度分割成子注释，对子注释进行一些处理后再合并为完整的父注释。本章对子注释处理部分进行阐述，分别包括语句粒度的注释克隆分类和注释分类。本章主要论述上述技术的基本思想，实现框架以及实现流程。对于每个分类和克隆级别的定义和实现也分别予以解释。

3.1　概述

本章阐述注释克隆检测技术中的子注释处理部分，主要包括数据清洗，子注释分类和子注释克隆分类三个步骤。子注释指的是将原始注释以语句粒度切分后得到的若干注释单元。

注释克隆检测技术采用的数据样本是代码-注释变更，每个代码-注释变更由一个四元组组成: 旧代码，新代码，旧代码对应的语句级注释，新代码对应的语句级注释。本章会按语句粒度对代码-注释变更样本进行切割，得到代码-子注释变更样本。为了让样本具备更多的语义信息，增强方法的可表示性，本章会对代码-子注释变更样本进行子注释分类和子注释克隆检测。而拥有完整注释的代码-注释变更样本的注释克隆检测在下一章进行。具体而言，假设一个注释由 5 个类型的子注释组成，子注释的克隆分为 4 个级别，更高级别的克隆意味着该类型的子注释在版本变更中的改动更小，因此，实际存在两个层面的注释克隆: 因此，实际存在两个层面的注释克隆:

1. 语句粒度的子注释的克隆.
2. 整体父注释的克隆.

本文的目标是检测出情况 (2), 即父注释的克隆。而在本章中，会检测出情况（1），即子注释的克隆，将该信息用于后续对情况（2）的检测。本章介绍所使用的数据清洗方法，注释分类方法, 以及对子注释的克隆分类和检测方法。

3.2　注释分类

软件开发者对代码的注释是多样的，他们倾向于从代码的不同方面进行注释。考虑如下两段注释:

1. “Copyright (c) 2010-2022 Apache Corporation.”

2. “Shifts any subsequent elements to the let (subtracts one from their indices).ž of method remove(int index). ”

前者描述了方法的实现细节，而后者描述的是代码的版权信息。对于方法代码的更新，前者需要同步地进行更新来维持代码-注释的不一致，而后者则不需要这么做。虽然在广义上，二者都能够看作是注释克隆，因为它们的注释的文本部分极为相似，但将第二个例子中的注释的相似显然对分析代码-注释的不一致性没什么意义。

考虑如下两个代码块，它们都带有相似的注释:

```
/**
 * Copyright 2004-2021 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 */
public void calculateSum(int a, int b) {
    int sum = a + b;
    System.out.println("The sum is: " + sum);
}
```

```
/**
 * Copyright 2004-2021 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 */
public void calculateSumAndProduct(int a, int b) {
    int sum = a + b;
    int product = a * b;

    if (sum > product) {
        System.out.println("The sum is greater than the product.");
    } else if (sum < product) {
        System.out.println("The product is greater than the sum.");
    } else {
        System.out.println("The sum and the product are equal.");
    }
}
```

可以看到，两个代码块的代码部分做出了很大的改动，但是注释部分是相似的。在很多情况下需要将这种情况识别为代码-注释的不一致，但是，对于这个例子中的情况，由于注释部分仅仅描述了代码的版权信息，因此注释部分的不变是正常的。从这个例子可以看到，代码的注释有不同种类，而不同种类的注释的相似在不同应用场景下有不同的意义。因此有必要对注释进行分类处理，以获得更好的效果。

Zhai 等人[19]提出的 CPC 是一个注释分类和注释传播分析工具。CPC 根据注释对应的代码粒度（例如类，方法和声明）以及注释描述的代码方面（例如代码的功能，代码的实现细节，代码的使用方式）进行分类。本文提出的注释克隆检测技术结合了 CPC 的分类法，本文只在方法层面进行检测。这个分类法的好处是它在处理注释的同时考虑了程序的语义，因此可以更好地揭示程序中的错误。CPC 在方法层面将注释分为五类: what，why，how-it-is-done，property，how-to-use。下面分别介绍这五种分类:

1. what：what 类型的注释提供了对应代码部分的定义或者功能的描述，可以从 what 注释中提取出关键的代码语义信息。例如代码的安全敏感性。这对漏洞识别很重要。通过阅读这种类型的注释，开发人员可以很容易地理解相应的代码实体的主要功能，而不需要潜入（实现）细节。例如，表 1 中第一行的注释"Pushes an item onto the top of this stack" 描述了方法 *push*（*E item*）的主要功能。
2. why：why 类型的注释提供了开发者提供代码部分的原因，或者设计该代码的原理。该类型的注释在两种情况下很重要。首先，它可以帮助开发者理解那些实现非常复杂而无法快速理解用途的方法。其次，如果存在多个看起来相似但目的不同的方法，开发者可以提供 why 类型注释来解释为什么要提供这些方法，并解释这些方法不是普通的冗余。
3. how-it-is-done：how-it-is-done 类型的注释描述了代码实现的细节。这些信息对于开发人员理解代码非常重要，特别是在代码复杂度很高的情况下。检测 how-it-is-done 类型的注释和代码实现之间的不一致是发现注释克隆和代码-注释不一致重要方式。
4. property：property 类型的注释描述了代码的属性，例如方法的前置条件和后置条件。前置条件规定了使用该方法时应该具备的条件，而后置条件则表明了使用该方法的结果。例如，对于方法 *setElementAt*（*E*，*int*）的一个参数 index，该方法的 property 类型注释 "The index must be a value greater than or equal to 0" 指明了一个前提条件 index≥0，必须满足这个条件，该方法才能正常工作。
5. how-to-use：how-to-use 类型的注释描述了使用对应代码所需的一些前提条件，例如平台和依赖库的版本。例如，抽象类 *AbstractIterator* 的注释 "But using this class，one must implement only the computeNext method，and invoke the endOf- Data method when appropriate。" 明确指出了其具体类中所需要的实现。这些注释对于代码注释的不一致性检测非常重要。[9]。

3.2.1 Zero-shot 注释分类器

本文使用 zero-shot learning[48]技术来实现 CPC 提出的注释分类方法。Zero-shot learning 是一种无监督的机器学习方法，允许在没有标记样本的情况下识别和分类新的未知类别。具体的数学过程如下：

给定一个已知类别集合 C 和它们的属性表示 A，以及一个未知类别集合 U，Zero-shot learning 的目标是学习一个函数 f，将属性表示 A 映射到类别集合 C。

为了达到这个目的，Zero-shot learning 使用训练数据 $D = (x_i, a_i, c_i)$，其中 x_i 是输入样本，a_i 是与样本 x_i 关联的属性表示，c_i 是样本 x_i 的类别标签。学习函数 f 通过最大化联合概率 $P(c, a \mid x)$ 来实现，其中 c 表示类别，a 表示属性，x 表示输入样本。

使用贝叶斯规则，可以将联合概率 $P(c, a \mid x)$ 分解为条件概率 $P(c \mid a, x)$ 和先验概率 $P(a \mid x)$ 的乘积：

$$P(c, a \mid x) = P(c \mid a, x) * P(a \mid x)$$

条件概率 $P(c \mid a, x)$ 表示在给定属性 a 和输入样本 x 的情况下，类别 c 的概率。先验概率 $P(a \mid x)$ 表示在给定输入样本 x 的情况下，属性 a 的概率。

通过最大化 $P(c, a \mid x)$，即选择具有最高条件概率 $P(c \mid a, x)$ 的类别 c 作为预测结果，zero-shot learning 可以将属性与类别关联起来，从而进行零样本分类。

本文使用了预训练的 zero-shot 模型作为分类器, 如图3.1。

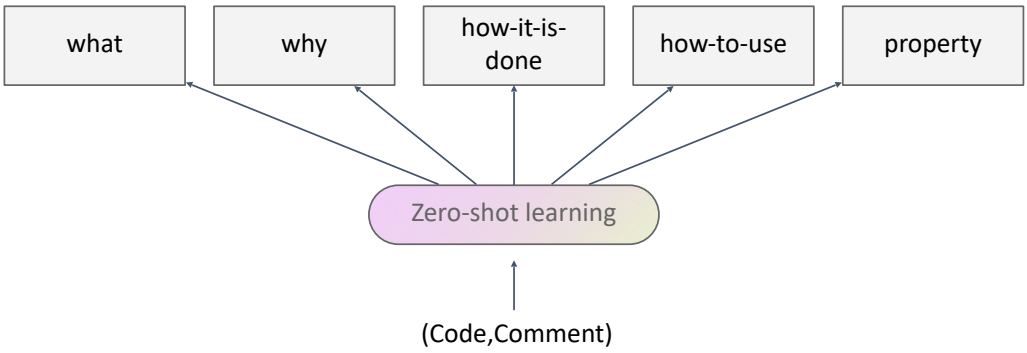

图 3.1 Zero-shot CPC 分类器

本文将代码-子注释变更样本的旧代码和旧注释部分提取出来，作为 zero-shot learning 的样本的属性表示 A，并使用 CPC 提出的五个分类作为 zero-shot 的类别集合 C，输入模型进行分类。

3.3 语句粒度的注释克隆分类

对于一个注释段，它由若干语句组成。本文在语句级别定义注释的克隆关系，并使用神经网络来学习注释克隆。在这一环节中参考代码克隆的 4 种类型来定义语句级别的注释克隆类型。本文将语句级别的注释克隆分为如下 4 类:

1. 类型 1（文本相似性）: 除了空空白字符之外完全相同的注释对。
2. 类型 2（词汇相似性）: 除了关键字（例如函数名，函数参数，类型名）之外都相同的注释对。

Algorithm 1 语句粒度的注释克隆分类算法

Input: Dataset A //由多个代码-注释更改样本组成的数据集 A
Output: Dataset A //数据集中的每个样本都被进行了注释克隆分类。

```
for sample in A do
    for cloneType in Textual, Lexical, Syntactical, Semantic, notClone do
        判断 sample 是否属于当前 cloneType
        if sample 属于当前 cloneType then
            Continue
        else
            //继续迭代，直到 sample 和某个 cloneType 匹配，或者匹配 notClone
        end if
    end for
end for
```

3. 类型 3（语法相似性）: 相比原来的注释，有若干语句的增删，或使用了不同的关键字，以及时间戳，证书，超链接，但是依然相似的注释对。
4. 类型 4（语义相似性）: 功能相同的注释，在文本或者语法上不相似，但在语义上有相似性。

算法1描述了对数据集进行语句粒度的注释克隆分类的流程。对每个代码-注释更改样本提取出其注释变更部分，也就是旧注释和新注释。本文使用自然语言处理技术来对每个注释变更进行克隆检测。本文规定，每个注释变更仅仅存在一种克隆分类，即不存在具有多个克隆级别的注释。最后，本文将子注释克隆的信息标注到原样本上。下面分别阐述四个级别的子注释克隆检测的实现方法，为了方便，使用 Type-1，Type-2，Type-3，Type-4 克隆来称呼第一，二，三，四级别的子注释克隆。

3.3.1 第一级别的注释克隆

对于 Type-1 克隆的检测是比较简单的，由于 Type-1 克隆的注释对仅仅在空白符号上有差异，因此只需要去除注释对中的注释的空白符号，比较两条注释的相似度即可。空白符号包括空格，换行符和制表符。

在相似度比较方面，本文使用 Python 的 difflib 库来实现。difflib 的 SequenceMatcher 类提供了比较两个文本字符串的方法，通过计算最长公共子序列的长度来确定两个序列之间的相似度，即相似部分的长度与序列总长度的比值。如果该相似度超过本文规定的阈值，那么就认为这对注释存在 Type-1 克隆。因此，Type-1 级别的克隆检测步骤为:

1. 去除空白符号。
2. 使用 difflib 的 SequenceMatcher 类进行比较。

3.3.2 第二级别的注释克隆

由于 Type-2 克隆包括了 Type-1 克隆，因此 Type-2 克隆检测的第一步就是检测输入的注释对是否满足 Type-1 克隆的情况，如果是，则认为是 Type-2 克隆，否则就进行下一步处理。Type-2 克隆的注释对是重命名的注释对，它描述了注释克隆对之间的词法差异。因此，本文将注释文本转换为词法单元（Token），比较两个词法单元序列的相似度。如果该相似度超过规定的阈值，则认为给定的注释对存在 Type-2 克隆。考虑到注释本身是自然语言，因此词法单元也就是注释中的每一个单词。本文使用 Levenshtein 距离来度量两个词法单元序列的相似程度。Levenshtein 距离是一种度量字符串之间差异的指标，它衡量通过插入、删除和替换操作将一个字符串转换为另一个字符串所需的最小操作次数，因此能够捕捉到文本之间的词法差异。具体来说，当计算 Levenshtein 距离时，它会考虑以下情况：

- 插入操作：当一个文本中插入一个字符来匹配另一个文本时，Levenshtein 距离会增加。
- 删除操作：当一个文本中删除一个字符来匹配另一个文本时，Levenshtein 距离会增加。
- 替换操作：当一个文本中替换一个字符以匹配另一个文本时，Levenshtein 距离会增加。

通过计算 Levenshtein 距离，可以得到一个表示两个文本差异程度的数值。如果该数值超过规定的阈值，就认为注释对存在 Type-2 克隆。

3.3.3 第三级别的注释克隆

和 Type-2 克隆类似，Type-3 克隆也包括了前两个级别克隆的情况。因此 Type-3 克隆检测的第一步就是判断注释对是否属于 Type-1 或 Type-2 克隆。Type-3 克隆发生在语法层面，对于语法层面的相似度分析有两个方法：基于树的相似度分析和基于度量的对比。本文使用 Stanford CoreNLP 库对注释进行基于树的语法分析，通过将文本转换为语法树并计算语法树的相似度来判定注释对是否相似。

和 Type-2 克隆类似，Type-3 克隆也包括了前两个级别克隆的情况。因此 Type-3 克隆检测的第一步就是判断注释对是否属于 Type-1 或 Type-2 克隆。Type-3 克隆发生在语法层面，对于语法层面的相似度分析有两个方法：基于树的相似度分析和基于度量的对比。本文使用 Stanford CoreNLP 库对注释进行基于树的语法分析，通过将文本转换为语法树并计算语法树的相似度来判定注释对是否相似。

具体而言，Stanford CoreNLP 库提供了方法将注释文本转化为语法解析树，它可以用于比较不同注释之间的句法结构相似性。解析树是由树节点和它们之

Algorithm 2 Type-3 克隆检测算法

Input: 由旧注释和新注释组成的注释对 (comment2, comment2) , 相似度阈值 threshold

Output: Boolean //表明该注释对是否具有 Type-3 克隆。

```
    if 属于 Type-1 克隆 then
2:      return True
    else if 属于 Type-2 克隆 then
4:      return True
    end if
6:  加载 Stanford CoreNLP 库, 检测语法相似度。
    totalScore = 0
8:  tokens1 = tokenize(comment1)
    tokens2 = tokenize(comment2)
10: trees1 = parse(tokens1)
    trees2 = parse(tokens2)
12: for  tree1 ∈ trees1 do
        for  tree2 ∈ trees2 do
14:         score = LevenshteinDistance(tree1, tree2)
            totalScore+ = score
16:     end for
    end for
18: avgScore = totalScore/(len(trees1) + len(trees2))
    if avgScore > threshold then
20:     return True
    else
22:     return False
    end if
```

间的关系构成的结构，每个节点代表一个短语或词汇，而边表示它们之间的语法关系。该结构可以表示文本中词汇之间的语法关系，包括主谓关系、动词宾语关系、修饰关系等。对于每个文本，本文会生成多个解析树，因为一个句子可以有多种解析方式。在得到解析树后，本文会使用嵌套的循环遍历每个解析树对，使用 Levenshtein 距离来计算解析树之间的相似程度。每个注释对都具有一个相似度得分，每次迭代时的解析树相似程度都会被加入到该得分。最后，本文将所有得分加起来并除以总对数，得到平均得分。这个平均得分代表了注释对在语法层面的相似度。

3.3.4 第四级别的注释克隆

Algorithm 3 Type-4 克隆检测算法

Input: 由旧注释和新注释组成的注释对 (comment2, comment2)，相似度阈值 threshold
Output: Boolean //表明该注释对是否具有 Type-4 克隆。
if 属于 Type-1 克隆 **then**
  return True
else if 属于 Type-2 克隆 **then**
  return True
else if 属于 Type-3 克隆 **then**
  return True
end if
加载 BERT 模型, 检测语义相似度。
tokens1 = tokenize(comment1)
tokens2 = tokenize(comment2)
embeddings1 = convertToEmbedding(tokens1)
embeddings2 = convertToEmbedding(tokens2)
similarity=cosineSimilarity(embeddings1, embeddings2)
if similarity > threshold **then**
  return True
else
  return False
end if

Type-4 克隆指的是在前三类克隆的基础上，注释的语义和功能相同。因此，Type-4 克隆的第一步是判断注释对是否属于前三类克隆。本文使用预训练的 BERT 模型来检测 Type-4 克隆。BERT 模型是一个基于 Transformer 的深度双向编码器，经过大规模的语言训练得到，通过对输入序列的全局建模和上下文理解，它在自然语言处理任务中表现出色，可以捕捉文本的语义信息。

具体而言，BERT 模型会在大规模无监督的文本语料上进行预训练，在预训练完成后，BERT 模型通过微调（fine-tuning）在特定任务上进行端到端的训练。BERT 模型在预训练阶段通过两个任务来学习文本表示：Masked Language Model

（MLM）和 Next Sentence Prediction（NSP）。输入的文本被分词并嵌入为词向量，同时加入特殊的标记（如 [CLS] 和 [SEP]）来表示句子的开头、结尾和分隔。经过多个编码器层的处理后，最后一个特殊标记 [CLS] 所对应的向量表示被用作整个输入序列的语义表示。BERT 模型由多个编码器层组成。每个编码器层包含多头自注意力机制（Multi-Head Self-Attention）和前馈神经网络（Feed-Forward Neural Networks）。自注意力机制允许 BERT 模型在编码过程中对输入序列中的所有位置进行联合建模，捕捉上下文相关的语义信息。下面介绍 BERT 模型的自注意力机制和编码器结构。

自注意力机制用于计算输入序列中每个位置的上下文相关表示。给定输入序列 $X = [x_1, x_2, \ldots, x_n]$，其中每个位置 x_i 表示输入的一个词语，自注意力机制计算每个位置的输出表示 $H = [h_1, h_2, \ldots, h_n]$。

1. 注意力权重计算：首先，通过将输入序列 X 映射为三个线性变换，得到查询（Query）矩阵 Q、键（Key）矩阵 K 和值（Value）矩阵 V。每个矩阵的维度为 d_{model}，与输入序列的词嵌入维度相同。然后，计算查询矩阵 Q 和键矩阵 K 之间的相似度得分，可以使用点积注意力或双线性注意力来计算相似度得分。得分矩阵通过 softmax 函数进行归一化，得到注意力权重矩阵 A，其中 A_ij 表示位置 i 与位置 j 的注意力权重。
2. 加权求和：将值矩阵 V 与注意力权重矩阵 A 相乘，得到加权求和后的表示。每个位置的上下文表示 h_i 是值矩阵 V 的加权求和，即 $h_i = \sum_j (A_{ij} V_j)$。
3. 多头机制：为了增强模型的表达能力，BERT 采用多头机制，将自注意力机制并行应用于不同的投影矩阵（Query、Key、Value）。具体地，BERT 将输入序列映射为多组不同的 Query、Key、Value 矩阵，然后分别计算每组的注意力权重矩阵和上下文表示。最后，将多个头的上下文表示拼接起来，并经过线性变换得到最终的表示。

BERT 模型由多个编码器层组成，每个编码器层包含两个子层：多头自注意力子层和前馈神经网络子层。

1. 多头自注意力子层：这个子层使用自注意力机制来对输入序列进行编码。在每个注意力头中，使用不同的投影矩阵来计算注意力权重和上下文表示。每个注意力头的输出被拼接在 BERT 模型的每个编码器层中，多个注意力头的输出被拼接在一起，并经过线性变换得到最终的上下文表示。
2. 前馈神经网络子层：在每个编码器层中，除了多头自注意力子层，还包含一个前馈神经网络子层。该子层由两个全连接层组成，中间使用 ReLU 激活函数进行非线性转换。

在每个编码器层中，多头自注意力子层和前馈神经网络子层都采用了残差连接（residual connection）和层归一化（layer normalization）的结构。这些机制有助于提高模型的梯度流和训练效率。

如算法3所示, Type-4 克隆检测的过程如下：首先，使用模型提供的词法分析器将注释对并分别地分词并转换为词法单元序列。其次，将词法序列转换为对应的嵌入向量。嵌入向量是一个具有固定长度的向量表示，它将文本的语义信息编码为向量空间中的点，实现对每个位置的词语进行上下文相关的表示。最后，本文使用余弦相似度，通过计算两个向量之间的夹角余弦值来衡量两个嵌入向量之间的相似程度。如果相似度超过了规定的阈值，则认为注释对具有 Type-4 克隆。

3.4 本章小结

本章介绍了注释克隆检测的思想和用到的主要技术：注释分类, 语句粒度的注释克隆分类和注释克隆. 注释分类和语句粒度的注释克隆分类用于子注释的处理部分，由于存在着描述代码的不同角度的注释, 注释克隆对它们的变更的容忍度也不尽相同，因此本文采取注释分类方式, 每个子注释拥有各自的分类，不同分类的子注释在注释克隆检测时被区别处理。此外，为了衡量子注释的相似程度, 本文参考代码克隆的四种分类，定义了语句粒度的注释克隆分类, 使用自然语言处理技术予以实现。

第四章　完整注释的克隆检测

在经过上一章的处理后，已经得到了处理完毕的代码-子注释变更样本，这些样本已经具备了子注释的分类和注释克隆信息。接下来需要聚合数据样本，将属于同一父注释的样本合并，并人工给合并后的代码-注释变更样本赋予标签。本章提出了完整注释的克隆检测规则，对样本进行人工标注的工作是建立在这些规则上的。最终，标注后的数据集被划分为训练集和测试集输入神经网络并进行模型训练。

本章介绍提出的完整注释克隆检测规则，对具有完整注释的代码-注释更改样本的标注过程，以及模型训练所使用的神经网络。此外，本章以 CUP2 数据集为例，介绍本文提出的注释克隆检测技术在大型 Java 代码库中的应用，阐述应用的背景，数据集优化过程和方法架构。由于实际的数据集的质量无法保证，可能存在一些噪声样本，对后续模型训练产生负面影响，因此本文还会对数据集进行样本清晰，本章在最后介绍数据集清洗的过程。

4.1　应用背景

代码-注释一致性关系被广泛地用在程序分析，错误检测，软件质量分析等领域。本文的注释克隆检测技术主要针对方法级别的 Java 程序，本文在 Liu 等人提出的 CUP2 数据集上进行方法的应用。

该数据集基于 Java 程序，包含了代码更改中修改的方法及其标头注释（方法注释）。这是因为 Java 方法可以与其标头注释精确关联，而准确链接注释和其他粒度的代码并非易事，例如，语句和方法注释是一种重要的注释类型。它们经常被开发人员参考以进行程序理解，并可用于构建 API 参考文档 [50]。此外，CUP2 在句子级别检测和更新过时的评论，即一次处理一个评论句子。这个选择的原因是：

1. 语句的粒度更小，识别起来更连贯也更容易[49]。
2. 具有多个句子的方法注释也可以迭代处理和更新。

该数据集由大约 4086K 数据样本组成。Liu 等人[24]从 GitHub 上克隆了 1496 个 Java 仓库，通过 Wen 等人[3]的人工验证确认这些仓库的真实性。它们最终提取了提取了 4357K+ 合格的代码-注释更改样本，每个样本是个五元组: 旧代码，新代码，旧注释，新注释，标签。其中标签部分表明了该样本是否具有过时注释。由于本文的研究目标是注释克隆，因此本文会舍弃原数据集中的标签字段。

所以，本文实际采用的数据样本是如下元素构成的四元组：旧代码，新代码，旧注释，新注释。其中的注释部分已经按照语句粒度进行了切割，也就是说 CUP2 数据集中的每个样本都已经是代码-子注释变更样本，再经过处理后，这些样本会被合并为代码-注释变更样本。

需要注意的是，Xu 等人[16]提出的 cleaned OCDDATA 同样基于 CUP2 数据集，并且做到了质量提升。但是他们的方法过滤掉了更多信息，这些信息对于后续的注释克隆分类等步骤可能是有用的。因此本文仍然使用 CUP2 数据集，而不是 Xu 等人提出的 cleaned OCDDATA。

4.2 注释克隆

在第三章中已经对子注释进行了处理，处理完毕的代码-子注释变更样本已经具备了子注释的分类和注释克隆信息。接下来需要聚合这些样本，将属于同一父注释的样本合并。然后对合并后的代码-注释变更样本进行注释克隆检测，标注者应用本文定义的注释克隆的规则，为每个样本赋予标签。本文规定样本的标签为阳性代表其存在注释克隆，阴性代表存在。所有样本的初始标签都是阴性，由标注者将符合规则描述的样本标注为阳性。

本节定义完整注释的克隆规则，对每个完整注释，建立其子注释的类型和子注释的克隆类型到该父注释的克隆情况的映射。本文会认为某些类型的子注释和某些类型的子注释克隆会和父注释的克隆有更明显的关联。具体而言，本文会对每个类型的子注释分配不同的权重，完整注释的克隆情况是由其每个分类的子注释所具有的权重，以及该类别子注释的克隆级别决定的。更高权重的子注释类别和更高的子注释克隆级别都会导致该父注释拥有更大的可能被认为存在克隆情况。完整注释克隆的定义如下：

- 如果注释中含有 property 类型的子注释，由于它描述了代码的属性，尤其是方法的前置和后置条件等信息，因此如果它具有词法级别的克隆，就认为该父注释存在注释克隆，赋予其最高的权重。
- 如果注释中含有 how-it-is-done 类型的子注释，由于它描述了代码的实现细节，对理解代码十分重要，也与代码的语义有很强的耦合，因此使用最严格的克隆级别。如果它具备文本级别的克隆，就认为该父注释存在注释克隆，赋予其较高的权重。
- 如果注释中含有 what 类型的子注释，由于它提供了方法的定义或者功能的描述，随着代码的更改，这些描述可能出现关键词的替换和增删。如果它具备语法级别的克隆，就认为该父注释存在注释克隆，赋予较高的权重。
- 如果以上原则都不能决定父注释的克隆情况，那么就观察 how-to-use 类型和 why 类型的子注释，前者描述了使用对应代码所需的一些前提条件，后者描述了开发者提供这些代码的原因，这些描述可能是高度自然语言化的，

因此，如果这两个类型的子注释存在语义级别的克隆，本文就认为该父注释存在克隆，赋予较高的权重。

通过上述规则的帮助，实验者对代码-注释变更样本进行人工标注。

4.3 人工标注

在之前的步骤中，代码-子注释变更样本已经经过了分类和克隆检测的处理，并且合并为具有完整注释的代码-注释变更样本。本文同样提出了针对完整注释的克隆检测规则。本节介绍如何应用这些规则对代码-注释变更样本进行人工标注，识别注释克隆样本。为了尽量减少主观性，所有标注者都具有至少四年的编程经验，并且对代码的注释比较熟悉。由于本文采用手工标注，不可避免地会引入主观性。为了减少这种主观性，标注方法使用打分机制。标注者会对每个样本按照制定的规则进行打分，如果分数高过一定阈值，则认为该样本存在注释克隆。

4.4 神经网络架构

在对数据集进行人工标注后，本文使用神经网络来对注释分类检测进行学习，本节介绍使用的神经网络，如图 4.1，本节介绍该网络的原理。

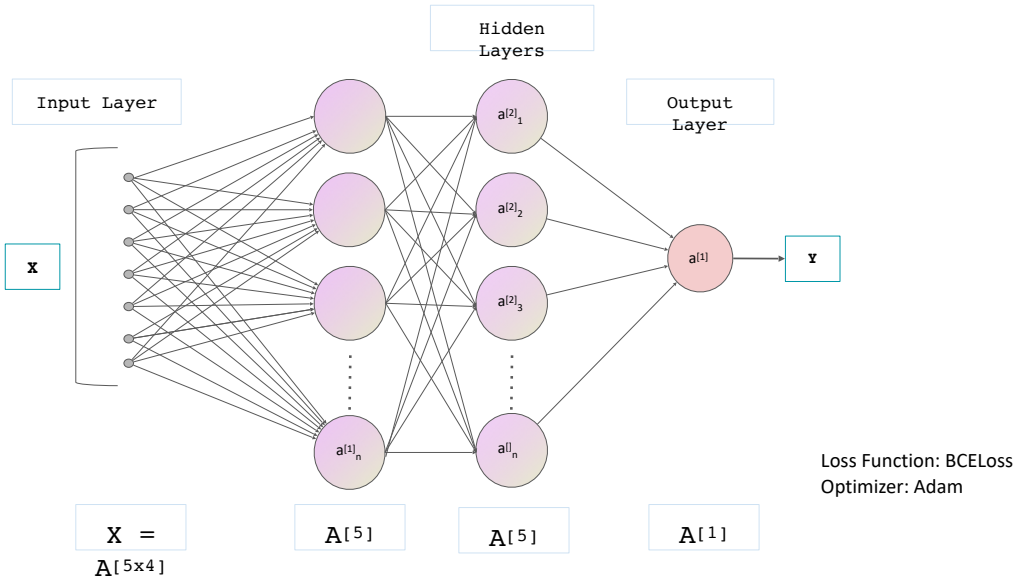

图 4.1 注释克隆检测技术使用的神经网络

在通过计算权重是否超过阈值并对子注释赋予标签后，再将原本属于同一父注释的子注释进行合并，得到合并后的代码-注释变更样本，以及其构成的数据集，将该数据集分为训练集和测试集，将训练集输入神经网络进行训练. 接下来介绍该网络的原理.

对于每个样本，由于它已经是合成后的样本，因此其注释部分是多个子注释合成的父注释，并且还具有各个子注释的分类和克隆信息。本文将每个样本转化为输入特征 x，这是一个 5×4 维的张量，对应子注释的五个类别和四个克隆等级。网络的输出是一个一维的张量 y，通过输出层的线性变换和激活函数得到，代表该样本是否存在注释克隆。y_{true} 是样本的真实标签。

本文的目标是使输出 y 尽可能接近真实标签 y_{true}。在模型训练过程中，通过计算损失函数的梯度来更新模型的参数，使用 Adam 优化器根据梯度和历史梯度平方的指数移动平均来调整参数的更新步长，从而逐步优化模型的性能。

本文使用二元交叉熵损失（Binary Cross Entropy Loss）函数来度量预测值 y 与真实标签 y_{true} 之间的差距，同时使用 Adam 优化器来更新神经网络中的参数。

本文的目标是使输出 y 尽可能接近真实标签 y_{true}。在模型训练过程中，通过计算损失函数的梯度来更新模型的参数，使用 Adam 优化器根据梯度和历史梯度平方的指数移动平均来调整参数的更新步长，从而逐步优化模型的性能。

本文使用二元交叉熵损失（Binary Cross Entropy Loss）函数来度量预测值 y 与真实标签 y_{true} 之间的差距，同时使用 Adam 优化器来更新神经网络中的参数。

本文的目标是使输出 y 尽可能接近真实标签 y_{true}。在模型训练过程中，通过计算损失函数的梯度来更新模型的参数，使用 Adam 优化器根据梯度和历史梯度平方的指数移动平均来调整参数的更新步长，从而逐步优化模型的性能。

本文使用二元交叉熵损失（Binary Cross Entropy Loss）函数来度量预测值 y 与真实标签 y_{true} 之间的差距，同时使用 Adam 优化器来更新神经网络中的参数。

二元交叉熵损失可以用以下公式表示：

$$BCELoss(o,t) = -\frac{1}{n}\sum_{i=1}^{n}(t_i \log(o_i) + (1-t_i)\log(1-o_i))$$

其中，o 代表模型的输出，t 代表实际的标签，n 代表样本的数量，t_i 和 o_i 分别表示第 i 个样本的实际标签和预测标签。实际标签为 1 时（这代表着网络将样本预测为阳性），损失函数惩罚模型输出概率 p 偏离 1 的程度；当实际标签为 0 时，损失函数惩罚模型输出概率 p 偏离 0 的程度。通过最小化该损失函数，模型可以学习将预测结果与实际标签尽可能地匹配。

Adam 优化器是一种基于梯度的优化算法，结合了动量（Momentum）和自适应学习率（Adaptive Learning Rate）的特性。它根据每个参数的梯度和历史梯度平方的指数移动平均来更新参数。其公式如下：

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

其中，θ_t 表示第 t 步的模型参数，η 是学习率，$\hat{m}_t$ 是梯度的一阶矩估计（动量），$\hat{v}_t$ 是梯度的二阶矩估计（RMSProp 算法的变种），ϵ 是一个很小的常数，用于数

值稳定性。Adam 优化器综合考虑了梯度的平均值和方差，可以自适应地调整学习率，并且能够有效地应对不同参数的梯度变化情况，从而提高优化的效果和收敛速度。

4.5 方法架构

注释克隆检测技术的架构分为两个部分，如图4.2和图4.3所示。

在图4.2中使用基于规则的方法进行数据集人工标注。具体而言，首先要对数据集进行数据清洗，去除重复样本和无效的样本。接下来，本文应用克隆检测技术，将该样本的子注释部分进行注释分类和子注释克隆检测，接着将属于同一个代码块的样本合并，进行注释克隆检测。这样本文就拥有了使用基于注释克隆定义进行标注的数据集。

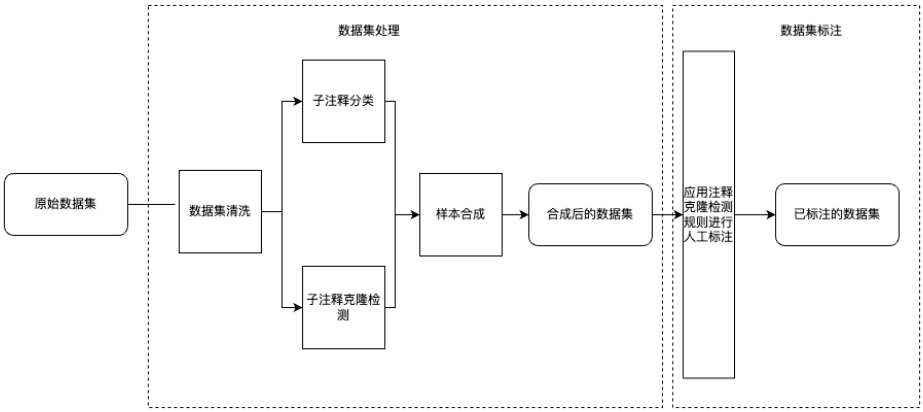

图 4.2 数据集处理和标注部分

拥有了标注好的数据集后就可以使用基于机器学习的方法训练模型，使用训练好的模型自动化检测注释克隆情况。如图4.3所示，首先按照时间顺序将数据集划分为训练集和测试集。接着，将训练集输入神经网络进行模型训练。最后，保存模型，使用测试集来评估模型。

对于新的代码-注释更改样本，可以按照图4.2的方式进行数据集处理，然后将处理后的数据集输入训练好的模型，进行图4.3中的模型评估环节，检测注释克隆的结果。

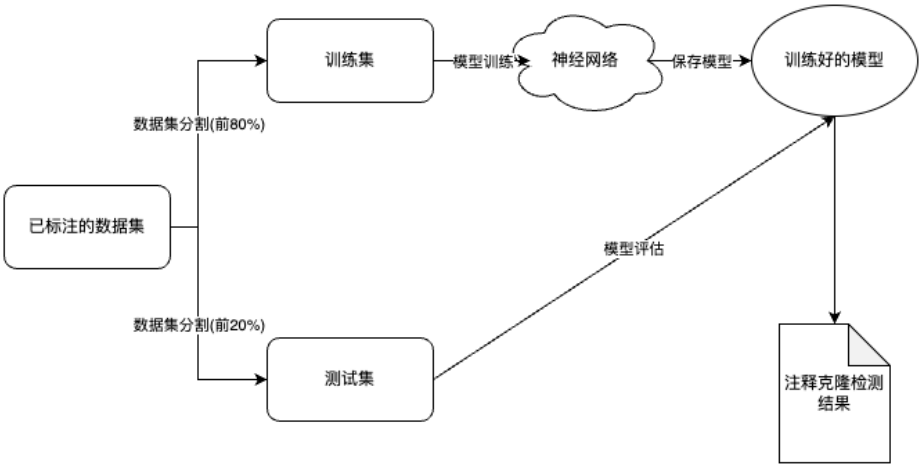

图 4.3 模型训练，评估和使用部分

4.6 数据集清洗

在这一部分，本文通过样本去重和信息过滤两个步骤来提高数据集质量，降低噪声样本对模型训练产生影响。由于这一步骤会删除样本，减少数据集的规模，因此数据集清洗步骤是在对代码-子注释变更进行处理前实施的。合并后的代码-注释变更样本不需要进行数据清洗。

- 样本去重：如果两个样本的输入基本相同（即相同的旧代码、新代码和旧注释），无论新注释是否相同，本文都认为它们是重复的。对于重复的样本，本文选择保留其中的一个随机样本。
- 信息过滤: 本文观察到，有些注释含有的信息是无效的，它们对模型训练部分而言属于噪声，因此需要去除这些注释。首先本文发现如果一段注释是行注释是行注释（即以”//” 开头的注释），而不是 Javadoc 或块注释，那么它通常是一个有注释的 Java 注解（例如，”//@Override”），而不是一个 Java 方法的描述。因此，本文首先删除了带有行注释的实例。接着，本文删除具有 *@inheritDoc*、代码片段和 HTML 标签的注释，以及含有非 ASCII 字符的注释。在这些操作之后，可能会出现空文档，本文进一步删除了有空文档的实例。最后，根据本文对 GumTree 计算的方法映射的检查，本文发现 GumTree 经常对抽象方法产生不准确的映射。因此，包含抽象方法的实例也被删除以减少方法的不匹配。

4.7 本章小结

在本章中阐述了如何对具有完整注释的代码-注释变更样本进行注释克隆检测的原则，并且以 CUP2 数据集为例，展示对大型代码库进行注释克隆检测方法的完整流程，这分为五部分:（1）对原始数据集进行清洗，提升数据集质量。(2) 对数据集中的每个代码-子注释变更样本进行进一步处理，对其应用子注释分类和子注释克隆检测方法。（3）将语句粒度的子注释合并。根据本章提出的注释克隆检测原则，对合并后的代码-完整注释变更进行人工标注。（4）标注后的样本被作为神经网络的数据集，将其输入神经网络，使用反向传播和梯度下降技术进行模型训练，得到最终的注释克隆检测器。(5) 对于新的样本，输入训练好的模型中，进行注释克隆检测。

第五章　工具实现与实例研究

本章本文介绍原型工具的实现，以及对该工具的评估结果。本文设计了不同的实验来评估该工具的准确性和有效性。

5.1　原型工具实现

本文在给定的实验环境下完成了注释克隆检测工具。实验环境如下：

1. 操作系统: Linux Ubuntu 20.04
2. 软件环境: Python = 3.8
3. CPU: 15 vCPU Intel（R）Xeon（R）Platinum 8338C CPU @ 2.60GHz
4. GPU: RTX 3090（24GB）* 1
5. 内存: 80GB
6. 硬盘: 100GB

本文采用 CUP2 作为原始数据集。数据集清洗模块负责对原始数据集的清洗，包括样本去重和标签纠正等步骤。接着，子注释分类和克隆检测模块对每个代码-子注释变更样本进行子注释的分类和克隆检测信息的标注，属于同一代码块的样本会在注释合并模块被合并为代码-注释变更样本。然后, 在注释克隆检测模块，由两个实验人员根据本文定义的注释克隆规则进行手动标注，判断该样本是否存在注释克隆。前 80% 的样本被作为训练集，后 20% 的样本被作为测试集。本文将训练集输入模型的训练, 测试集用作模型的评估。本文的注释克隆检测工具是高度自动化的, 用户可以将新的样本输入预测模块进行注释克隆检测。

5.2　实际程序实验

本节希望实验能够回答以下几个问题：

- Q1: 注释克隆检测技术的准确性如何?
- Q2: 注释克隆检测技术的有效性如何?
- Q3: 方法采用的子注释分类和子注释克隆检测技术对实验效果是否有提升?

为了回答上述问题，对注释克隆检测工具做出有效评估，本文使用了三个常用的评估指标：Precision（精确率）、Recall（召回率）、F1 Score。

- 精确率（Precision）：精确率是指模型预测为正例的样本中实际为正例的比例。在代码-注释一致性检测中，精确率表示正确预测为一致的代码注释对占所有预测为一致的代码注释对的比例。
- 召回率（Recall）：召回率是指实际为正例的样本中被模型正确预测为正例的比例。在代码-注释一致性检测中，召回率表示正确预测为一致的代码注释对占所有实际为一致的代码注释对的比例。
- F1 Score：F1 Score 是精确率和召回率的调和平均值，用于综合评估模型的性能。F1 Score 综合考虑了精确率和召回率，对于不平衡数据集有较好的效果。

由于实验使用的 GPU 内存的的限制，本文没有使用完整的 CUP2 数据集，而是截取了该数据集的 0.5%。该数据集已经将样本按照时间进行排序并划分为了训练集和测试集。实验使用的数据集和 CUP2 原始数据集的对比如表5.1。本文的训练集包含 15974 个样本，测试集包含 2271 个样本。在数据集清洗之后，训练集包含 15942 个样本，测试集包含 2267 个样本。接下来，本文规定训练集和测试集的样本数量比例为 4：1，即训练集中的样本数量为总数据集中的 80%，如果该比例不能满足，则去除多余的测试集样本，直到比例达到 80% 为止。最终，实验使用的训练集包含 9068 个样本，训练集包含 2267 个样本。

表 5.1 实验使用的数据集与原始数据集的对比

	CUP2 数据集	**实验使用的原始数据集**	**清洗过后的数据集**	**实验使用的数据集**
训练集	3194930	15974	15942	9068
测试集	454383	2271	2267	2267

数据集人工标注的结果见表5.2。为了减少主观性，本文采用两个标注者。所有编码员都有至少四年的编程经验并且熟悉代码注释。在标注开始前，一位标注者对数据集中随机抽取的 100 条样本进行了试点研究，确定了对于注释克隆的基本观点。接下来，根据本文定义的注释克隆原则，该标注者举行了 20 分钟的回忆，对其他标注者进行培训。在会议期间他们讨论了注释克隆的含义和例子，并澄清了出现的一些误解。之后，来自训练集和测试集的 11335 个样本被分配给所有标注者。为了减少主观性，所有标注者对同一份样本集合进行标注，并在结束后对出现分歧的标注开会讨论。

从表5.2可以看到，人工标注后的结果显示实验采用的数据集是不平衡的，数据集中的阳性样本远少于阴性样本。在训练集中阴性样本占总数据集大小的 96.5%, 在测试集中阴性样本占总数据集大小的 97.5%。

模型在测试集上的表现见表5.3 和表5.4。

表 5.2 实验使用的数据集的不平衡性

	阳性样本数	阴性样本数	阴性样本占总样本数的比例
训练集	319	8749	96.5%
测试集	56	2211	97.5%

表 5.3 测试集上的各项指标结果

	Precision	Recall	F1-Score	AUPRC
实验结果	97.6%	75.0%	84.8%	88.7%
不使用子注释克隆检测	94.0%	54.0%	68.0%	79.0%
不使用子注释分类	96.4%	43.5%	60.0%	73.3%

本文使用了CPC提出的注释分类法,将注释分成5类。使用zero-shot learning技术进行注释分类的实现。采用的预训练模型为“facebook/bart-large-mnli”[1]。实验得到的子注释分类结果如表5.5。

考虑到每个子注释可能同时具有多个分类，表5.6还统计了具有多个分类的子注释的数量。实验中没有发现具有 0 个分类或具有全部 5 个分类的样本。

子注释的克隆检测结果见表5.7，可以看到大部分的子注释都处于 Type-1 克隆，Type-3 克隆的数量低于 Type-1 克隆。而 Type-2 克隆和 Type-4 克隆的数量都比较少。

结论 1: 本文的注释克隆检测工具的准确性高。从表 1 可以看到，本文的方法拥有接近 96% 的准确率，99% 的召回率，可以有效地检测出注释克隆的情况。同时，考虑到本文采用的数据集中正负样本的不平衡性，本文还计算了精度-召回曲线下的面积（AUPRC）[32]，这是一个常用的于不平衡数据集的指标，显示了随着决策阈值的变化，精度和召回率会发生的变化。本文的 AUPRC 值达到了是 75.9%，表明本文的方法在不平衡的数据集上依然取得了很好的效果。

结论 2: 通过对本文使用的数据集和注释分类，克隆检测方法的调查，可以确认本文方法拥有较高的有效性。本文根据 CPC 实现了子注释分类方法，CPC 允许同一条注释拥有多个分类，但一个注释不能拥有全部的分类，因为这意味着分类法的失效。CPC 是一个完备的分类方法，这意味着所有注释都会拥有至少一个分类，分类结果从表5.6可以看到，实验证实了无论是训练集还是测试集，都不存在 0 个分类或者具有全部 5 个分类的情况，证明了本文采用的分类法的有效

[1]https://huggingface.co/facebook/bart-large-mnli

表 5.4 测试集上的预测结果

	TP	FP	TN	FN
实验结果	42	1	2210	14
不使用子注释克隆检测	31	2	2207	27
不使用子注释分类	27	1	2204	35

表 5.5 子注释分类结果

	what	why	how-it-is-done	property	how-to-use
训练集	2452	447	3646	7301	1361
测试集	675	108	904	1793	353

表 5.6 具备多个分类的子注释统计

	1 个类别	2 个类别	3 个类别	4 个类别
训练集	4345	3457	1133	133
测试集	1075	888	270	34

性。此外，在分类结果中，property 类型的数量大于 how-ot-is-done，how-it-is-done 的数量又大于 what，这和 CPC 论文中的实验结果是符合的。

结论 3: 本文采用了子注释分类和子注释克隆检测技术，目的是对每个数据样本增加额外的语义信息，加强模型效果。同时，使用这两个技术可以提升方法的可解释性。为了研究这两个技术的使用是否真的对实验效果有提升，在本章中还进行了对比实验。即分别进行了三次实验，这三次实验有如下的区别：（1）同时使用了子注释分类和子注释克隆检测技术。(2) 只使用了子注释分类技术。(3) 只使用了子注释克隆检测技术。实验结果见表5.3 和表5.4。可以看到，在去除这两个技术之一后，实验结果的各种度量均有所下降。在情况（2）中，precision 下降了 3.6%, recall 下降了 21%。在情况（3）中，precision 下降了 1.2%, recall 下降了 32%。实验结果表明，子注释分类和子注释克隆检测技术都对本文的方法的效果有提升。

5.3 实验讨论

本文的方法也存在几点限制:

1. 本文的克隆检测技术采用了一般的卷积网络，容易出现过拟合情况。
2. 本文采用的数据集是不平衡的，尽管通过计算 'AUPRC'指标确认了方法的有效性，但没有采取措施改变这种不平衡情况。
3. 本文的方法主要处理 Java 语言的方法级别注释，没有考虑其它代码块粒度和其它语言的情况。

限制 3 是因为所使用的 CUP2 数据集的限制。Liu 等人在 CUP2 的论文中声称，它们构造数据集的方法可以被迁移到其它语言的代码库中，并且本文的方

表 5.7 子注释克隆检测结果

	Type-1	Type-2	Type-3	Type-4
训练集	8765	38	113	76
测试集	2280	8	17	8

法的实现中也不存在与 Java 的语言特性强烈耦合的情况。此外，本文使用的注释分类方法 CPC 也可以处理其它代码块粒度的注释，因此，限制 3 对本文方法的有效性造成的威胁是最小的。实验数据表明本文的方法拥有很高的准确率，但是，本文的方法仅在有限的数据集上进行了评估，因此不能声称对所有的实际程序都具有如此高的准确性。

5.4 本章小结

本章阐述了克隆检测技术的实现，以及在 CUP2 数据集上的实验结果，在此基础上对本文提出的方法进行了评估。为了从多个角度对方法进行评估，本章中使用了多个实验指标。为了论证方法的有效性和准确性，本章中列举了一些关于本方法的问题，并且用实验结果予以回答。本章的实验主要包括两部分：准确性验证部分和方法可靠性验证部分。在准确性验证部分中，使用多个指标来论证了克隆检测的准确性，在方法可靠性验证部分，通过列举实验参数以及解释实验结果，论证了方法的有效性。

第六章　总结与展望

本章总结全文，并讨论后续工作。

6.1　总结

在现代软件开发中，注释在提供代码功能和设计意图解释方面起着重要作用。然而，当代码发生变化时，开发人员往往忽视相应注释的更新，导致注释不再与代码保持一致。过时的注释可能引入潜在的错误。这种不一致性可能会误导开发人员并降低软件质量。目前的研究主要集中在代码克隆检测和过时注释检测与更新两个方面，但各自存在一些问题。传统的代码克隆检测方法只关注代码部分，而忽视了注释部分，无法有效解决注释-代码不一致的问题。基于规则的模型采用数据流分析和人为定义的规则，能够结合代码和注释特征，但在复杂情况下可能不够有效。基于机器学习的方法使用神经网络，具有较强的表达能力和灵活性，但受数据集质量和缺乏可解释性的影响。本文对代码-注释一致性检测提出了一个全新的解决方案，即注释克隆检测。本文的贡献有以下几个方面：

1. 本文提出了注释克隆的定义，并实现了注释克隆检测技术。在方法的实现中，结合了注释分类，代码克隆，自然语言处理技术和神经网络，并通过神经网络训练得到的模型来自动化整个注释克隆检测流程。通过使用注释分类技术，本文提出的方法在处理注释时可以有效地结合代码的语义。通过定义注释克隆的规则，本方法相比传统的基于机器学习的方法具备更良好的可解释性。同时，通过使用神经网络技术，本方法拥有更强的泛化能力，可以处理大型代码库中的各种复杂情况。
2. 本文将提出的注释克隆检测技术应用在大型 Java 程序数据集 CUP2 上，并且阐述了技术应用的流程和细节。为了提高方法的效果，本文对原始数据集进行了数据集清洗工作，去除了噪声样本。同时本文也展示了从数据集清洗到子注释分类，子注释克隆检测，注释合并和模型训练的整个工作流程。
3. 本文进行了原形工具的实现以及评估工作，从多个角度论证方法的准确性和有效性。主要从以下三个角度: 采用各种度量来评估方法的准确性。通过对实验数据的解释来说明方法具有足够的有效性。

6.2 展望

未来的研究工作可以考虑以下几个方面:

1. 本文提出的注释克隆检测工具只处理方法级别的 Java 代码块，然而由于本文使用的方法对于具体的语言和代码块粒度并没有强烈的耦合，因此可以将方法扩展到其它语言，例如 C 语言和 Python，以及其他粒度的代码块，例如类级别，文件级别和行级别。
2. 本文处理的注释仅仅包括最一般的注释，即注释的描述部分。对于某些文档类的注释，还可能存在结构化的标签部分。以 Javadoc 为例，Javadoc 注释就存在描述和标签两部分，其中的标签部分可以用基于规则的方法处理。因此，本文后续可以采用数据流分析等技术，进一步处理这些结构化的标签注释，增加方法的泛化能力。
3. 本文采用的神经网络使用了二元交叉熵损失（Binary Cross Entropy Loss）函数和 Adam 优化器进行学习，后续可以选择其它的网络组件进行训练，并且和当前使用的组件对比，查看不同的神经网络组件对于网络训练的效果。同时，其他的神经网络结构也可以被采纳。考虑到注释具有自然语言结构，使用双向长段时记忆网络（LSTM）进行注释处理也许是一个不错的选择。

参考文献

[1] STEIDL D, HUMMEL B, JÜRGENS E. Quality analysis of source code comments[C]//IEEE 21st International Conference on Program Comprehension, ICPC 2013, San Francisco, CA, USA, 20-21 May, 2013. IEEE Computer Society, 2013: 83-92. DOI: 10.1109/ICPC.2013.6613836.

[2] VÁSQUEZ M L, LI B, VENDOME C, et al. How do Developers Document Database Usages in Source Code? (N)[C]//COHEN M B, GRUNSKE L, WHALEN M. 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015. IEEE Computer Society, 2015: 36-41. DOI: 10.1109/ASE.2015.67.

[3] WEN F, NAGY C, BAVOTA G, et al. A large-scale empirical study on code-comment inconsistencies[C]//GUÉHÉNEUC Y, KHOMH F, SARRO F. Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019. IEEE / ACM, 2019: 53-64. DOI: 10.1109/ICPC.2019.00019.

[4] PADIOLEAU Y, TAN L, ZHOU Y. Listening to programmers - Taxonomies and characteristics of comments in operating system code[C]//31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings. IEEE, 2009: 331-341. DOI: 10.1109/ICSE.2009.5070533.

[5] TENNY T. Procedures and comments vs. the banker's algorithm[J]. ACM SIGCSE Bull., 1985, 17(3): 44-53. DOI: 10.1145/382208.382523.

[6] TENNY T. Program Readability: Procedures Versus Comments[J]. IEEE Trans. Software Eng., 1988, 14(9): 1271-1279. DOI: 10.1109/32.6171.

[7] JIANG Z M, HASSAN A E. Examining the evolution of code comments in PostgreSQL[C]//DIEHL S, GALL H C, HASSAN A E. Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR 2006, Shanghai, China, May 22-23, 2006. ACM, 2006: 179-180. DOI: 10.1145/1137983.1138030.

[8] RUBIO-GONZÁLEZ C, LIBLIT B. Expect the unexpected: error code mismatches between documentation and the real world[C]//LERNER S, ROUNTEV A. Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Pro-

gram Analysis for Software Tools and Engineering, PASTE'10, Toronto, Ontario, Canada, June 5-6, 2010. ACM, 2010: 73-80. DOI: 10.1145/1806672.1806687.

[9] TAN L, YUAN D, KRISHNA G, et al. /*icomment: bugs or bad comments?*/[C] / /BRESSOUD T C, KAASHOEK M F. Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007. ACM, 2007: 145-158. DOI: 10.1145/1294261.1294276.

[10] TAN S H, MARINOV D, TAN L, et al. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies[C] / /ANTONIOL G, BERTOLINO A, LABICHE Y. Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012. IEEE Computer Society, 2012: 260-269. DOI: 10.1109/ICST.2012.106.

[11] BLASI A, GOFFI A, KUZNETSOV K, et al. Translating code comments to procedure specifications[C] / /. 2018: 242-253. DOI: 10.1145/3213846.3213872.

[12] ZHONG H, ZHANG L, XIE T, et al. Inferring Resource Specifications from Natural Language API Documentation[C] / /2009 IEEE/ACM International Conference on Automated Software Engineering. 2009: 307-318. DOI: 10.1109/ASE.2009.94.

[13] PANDITA R, XIAO X, ZHONG H, et al. Inferring method specifications from natural language API descriptions[C] / /2012 34th International Conference on Software Engineering (ICSE). 2012: 815-825. DOI: 10.1109/ICSE.2012.6227137.

[14] GOFFI A, GORLA A, ERNST M D, et al. Automatic generation of oracles for exceptional behaviors[C] / /ZELLER A, ROYCHOUDHURY A. Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016. ACM, 2016: 213-224. DOI: 10.1145/2931037.2931061.

[15] WONG E, ZHANG L, WANG S, et al. DASE: Document-Assisted Symbolic Execution for Improving Automated Software Testing[C] / /BERTOLINO A, CANFORA G, ELBAUM S G. 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1. IEEE Computer Society, 2015: 620-631. DOI: 10.1109/ICSE.2015.78.

[16] WU X, ZHENG W, XIA X, et al. Data Quality Matters: A Case Study on Data Label Correctness for Security Bug Report Prediction[J]. IEEE Transactions on Software Engineering, 2022, 48(07): 2541-2556. DOI: 10.1109/TSE.2021.3063727.

[17] PASCARELLA L, BACCHELLI A. Classifying code comments in Java open-source software systems[C]//GONZÁLEZ-BARAHONA J M, HINDLE A, TAN L. Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017. IEEE Computer Society, 2017: 227-237. DOI: 10.1109/MSR.2017.63.

[18] PASCARELLA L. Classifying code comments in Java mobile applications[C]//JULIEN C, LEWIS G A, SEGALL I. Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2018, Gothenburg, Sweden, May 27 - 28, 2018. ACM, 2018: 39-40. DOI: 10.1145/3197231.3198444.

[19] ZHAI J, XU X, SHI Y, et al. CPC: automatically classifying and propagating natural language comments via program analysis[C]//ROTHERMEL G, BAE D. ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020. ACM, 2020: 1359-1371. DOI: 10.1145/3377811.3380427.

[20] IBRAHIM W M, BETTENBURG N, ADAMS B, et al. On the relationship between comment update practices and Software Bugs[J]. J. Syst. Softw., 2012, 85(10): 2293-2304. DOI: 10.1016/j.jss.2011.09.019.

[21] FLURI B, WÜRSCH M, GALL H C. Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes[C]//14th Working Conference on Reverse Engineering (WCRE 2007), 28-31 October 2007, Vancouver, BC, Canada. IEEE Computer Society, 2007: 70-79. DOI: 10.1109/WCRE.2007.21.

[22] FLURI B, WÜRSCH M, GIGER E, et al. Analyzing the co-evolution of comments and source code[J]. Softw. Qual. J., 2009, 17(4): 367-394. DOI: 10.1007/s11219-009-9075-x.

[23] RATOL I K, ROBILLARD M P. Detecting fragile comments[C]//ROSU G, PENTA M D, NGUYEN T N. Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017. IEEE Computer Society, 2017: 112-122. DOI: 10.1109/ASE.2017.8115624.

[24] LIU Z, XIA X, LO D, et al. Just-In-Time Obsolete Comment Detection and Update[J]. IEEE Trans. Software Eng., 2023, 49(1): 1-23. DOI: 10.1109/TSE.2021.3138909.

[25] PANTHAPLACKEL S, LI J J, GLIGORIC M, et al. Deep Just-In-Time Inconsistency Detection Between Comments and Source Code[C]//Thirty-Fifth AAAI

Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021. AAAI Press, 2021: 427-435.

[26] LIU Z, CHEN H, CHEN X, et al. Automatic Detection of Outdated Comments During Code Changes[C]//2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC): vol. 01. 2018: 154-163. DOI: 10.1109/COMPSAC.2018.00028.

[27] TAN L, YUAN D, ZHOU Y. HotComments: How to Make Program Comments More Useful?[C]//HUNT G C. Proceedings of HotOS'07: 11th Workshop on Hot Topics in Operating Systems, May 7-9, 2005, San Diego, California, USA. USENIX Association, 2007.

[28] TAN L, ZHOU Y, PADIOLEAU Y. aComment: mining annotations from comments and code to detect interrupt related concurrency bugs[C]//TAYLOR R N, GALL H C, MEDVIDOVIC N. Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011. ACM, 2011: 11-20. DOI: 10.1145/1985793.1985796.

[29] SRIDHARA G. Automatically Detecting the Up-To-Date Status of ToDo Comments in Java Programs[C]//SARKAR S, SUREKA A, COTRONEO D, et al. Proceedings of the 9th India Software Engineering Conference, Goa, India, February 18-20, 2016. ACM, 2016: 16-25. DOI: 10.1145/2856636.2856638.

[30] MALIK H, CHOWDHURY I, TSOU H, et al. Understanding the rationale for updating a function's comment[C]//24th IEEE International Conference on Software Maintenance (ICSM 2008), September 28 - October 4, 2008, Beijing, China. IEEE Computer Society, 2008: 167-176. DOI: 10.1109/ICSM.2008.4658065.

[31] LIU Z, CHEN H, CHEN X, et al. Automatic Detection of Outdated Comments During Code Changes[C]//REISMAN S, AHAMED S I, DEMARTINI C, et al. 2018 IEEE 42nd Annual Computer Software and Applications Conference, COMPSAC 2018, Tokyo, Japan, 23-27 July 2018, Volume 1. IEEE Computer Society, 2018: 154-163. DOI: 10.1109/COMPSAC.2018.00028.

[32] MAALEJ W, ROBILLARD M P. Patterns of Knowledge in API Reference Documentation[C]//HASSELBRING W, EHMKE N C. LNI: Software Engineering 2014, Fachtagung des GI-Fachbereichs Softwaretechnik, 25. Februar - 28. Februar 2014, Kiel, Germany: vol. P-227. GI, 2014: 29.

[33] KUMAR N, DEVANBU P T. OntoCat: Automatically categorizing knowledge in API Documentation[J]. CoRR, 2016, abs/1607.07602.

[34] HAOUARI D, SAHRAOUI H A, LANGLAIS P. How Good is Your Comment? A Study of Comments in Java Programs[C]//Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement, ESEM 2011, Banff, AB, Canada, September 22-23, 2011. IEEE Computer Society, 2011: 137-146. DOI: 10.1109/ESEM.2011.22.

[35] AL-KASWAN A, IZADI M, van DEURSEN A. STACC: Code Comment Classification using SentenceTransformers[J]. CoRR, 2023, abs/2302.13149. DOI: 10.48550/arXiv.2302.13149.

[36] SHENEAMER A, KALITA J K. A Survey of Software Clone Detection Techniques[J]. International Journal of Computer Applications, 2016, 137: 1-21.

[37] SAJNANI H, SAINI V, ROY C K, et al. SourcererCC: Scalable and Accurate Clone Detection[G]//INOUE K, ROY C K. Code Clone Analysis. Springer Singapore, 2021: 51-62. DOI: 10.1007/978-981-16-1927-4_4.

[38] CHOI E, YOSHIDA N, ISHIO T, et al. Extracting code clones for refactoring using combinations of clone metrics[C]//CORDY J R, INOUE K, JARZABEK S, et al. Proceeding of the 5th ICSE International Workshop on Software Clones, IWSC 2011, Waikiki, Honolulu, HI, USA, May 23, 2011. ACM, 2011: 7-13. DOI: 10.1145/1985404.1985407.

[39] COSMA G, JOY M. An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis[J]. IEEE Trans. Computers, 2012, 61(3): 379-394. DOI: 10.1109/TC.2011.223.

[40] De PAULO SOBRINHO E V, LUCIA A D, de ALMEIDA MAIA M. A Systematic Literature Review on Bad Smells-5 W's: Which, When, What, Who, Where [J]. IEEE Trans. Software Eng., 2021, 47(1): 17-66. DOI: 10.1109/TSE.2018.2880977.

[41] KAMIYA T. CCFinderX: An Interactive Code Clone Analysis Environment[G]//INOUE K, ROY C K. Code Clone Analysis. Springer Singapore, 2021: 31-44. DOI: 10.1007/978-981-16-1927-4_2.

[42] JIANG L, MISHERGHI G, SU Z, et al. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones[C]//29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007. IEEE Computer Society, 2007: 96-105. DOI: 10.1109/ICSE.2007.30.

[43] WU Y, ZOU D, DOU S, et al. SCDetector: Software Functional Clone Detection Based on Semantic Tokens Analysis[C]//35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020. IEEE, 2020: 821-833. DOI: 10.1145/3324884.3416562.

[44] ZHANG J, WANG X, ZHANG H, et al. A novel neural source code representation based on abstract syntax tree[C]//ATLEE J M, BULTAN T, WHITTLE J. Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019. IEEE / ACM, 2019: 783-794. DOI: 10.1109/ICSE.2019.00086.

[45] WU M, WANG P, YIN K, et al. LVMapper: A Large-Variance Clone Detector Using Sequencing Alignment Approach[J]. IEEE Access, 2020, 8: 27986-27997. DOI: 10.1109/ACCESS.2020.2971545.

[46] WANG P, SVAJLENKO J, WU Y, et al. CCAligner: a token based large-gap clone detector[C]//CHAUDRON M, CRNKOVIC I, CHECHIK M, et al. Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. ACM, 2018: 1066-1077. DOI: 10.1145/3180155.3180179.

[47] ROY C K, CORDY J R. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization[C]// KRIKHAAR R L, LÄMMEL R, VERHOEF C. The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008. IEEE Computer Society, 2008: 172-181. DOI: 10.1109/ICPC.2008.41.

[48] YIN W, HAY J, ROTH D. Benchmarking Zero-shot Text Classification: Datasets, Evaluation and Entailment Approach[C]//INUI K, JIANG J, NG V, et al. Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019. Association for Computational Linguistics, 2019: 3912-3921. DOI: 10.18653/v1/D19-1404.

[49] YIN P, NEUBIG G, ALLAMANIS M, et al. Learning to Represent Edits[C]//7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net, 2019.

致　　谢

> "I think it had whispered to him things about himself which he did not know, things of which he had no conception till he took counsel with this great solitude - and the whisper had proved irresistibly fascinating. It echoed loudly within him because he was hollow at the core."
>
> — Joseph Conrad, *Heart of Darkness*

2019 年我来到南京大学社会科学系，幻想着通过转专业进入文学院，成为一名作家。那个时候的我对未来坚定不移，觉得我的人生是有使命的，觉得前方的某处一定有某件事等着我完成。一年后我在转专业面试时被拒绝，理由是 "我们不培养作家"。

这件事彻底地改变了我的一生，心灰意冷的我在 2021 年转专业进入软件学院，准备在这个 "枯燥" 的领域度过此生，但我很快发现了计算机的乐趣，并在此后的三年沉迷其中，无法自拔。并且由于计算机的就业前景好，，我也有底气要求家庭支持我出国。如果我在文学院，这些都不可能发生。

平心而论，我觉得我对计算机的投入已经到了无以复加的地步，等我回过神时已经快要毕业了，时间慢慢流过，没有发出任何声响。这三年可以用一句话概括: "I can do all things"，这不是赞美，一个尝试做所有事的人可以成为优秀的工程师，却成为不了科学家，科学家需要把时间专注在关键的问题上。我被折腾各种配置和工具带来的胜利感迷惑，却没有意识到真正重要的东西是什么。

我一直不敢坦诚地面对自己，我在计算机投入那么多的精力是因为我真的喜欢它胜过任何事，还是因为我只是想通过忙碌来逃避内心的愧疚感? 计算机对理性和专注的要求以及长期的自闭生活改变了我的性格，与此同时康拉德 "黑暗之心" 的意象不断地出现在我的脑海中。我想象自己 penetrate deeper and deeper into the heart of darkness，想象头顶的云层越来越厚，黑暗笼罩世界，心底景观慢慢被淹没。好在这种黑暗的状态不仅麻木了感官，也麻痹了我对岁月的察觉，因此我还拥有热情和力量。我把科研和去美国当作抵抗黑暗的手段。我知道人生是很艰难的，我没有多少选择，一步错步步错，一不小心就会失败。希望我能做出正确的选择。

最后，感谢我的导师潘敏学教授和翟娟教授，感谢他们对于我的科研启蒙和培养，我知道我的工作做得还不够，因为我还没有正确地转变到科研思维，我会努力去弥补这一点。感谢这篇论文相关工作中引用的各位作者，感谢所有在本研究中提供支持和帮助的人。