Eindhoven University of Technology

MASTER

Code and Comment Consistency Classification with Large Language Models

Elmers, Peter O.

Award date:
2023

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 09. Oct. 2026

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Data Mining Research Group

Code and Comment Consistency Classification with Large Language Models

Master's Thesis

Peter Elmers

Supervisors:
Prof. Dr. Mykola Pechenizkiy
Ir. Zeno M. van Cauter
Prof. Dr. Alexander Serebrenik

Final version

Eindhoven, September 2023

Abstract

Comments in software code are intended to help developers understand the code. In contrast to code itself which is executed by a machine, comments are primarily written for humans. Previous work has shown that code understanding dominates a professional developer's time, which emphasizes the importance of studying comments. The study of code comment consistency involves assessing whether written comments are consistent with the code they are associated with. In this work, we apply large language models to the problem of detecting inconsistent comments. Our project addresses a major gap in previous work, with 90% of previous studies focusing on Java. In reproducing past results, we also discover a significant annotation artifact that likely caused misleading results in the past. We find that a simple classifier which predicts only based on the number of newlines at the end of the example would achieve 80% accuracy.

To extend beyond previous work, we build a novel dataset of more than 80,000 examples of code and comment pairs across 4 programming languages: Java, Python, Go, and JavaScript. We find that fine-tuned versions of CodeBERT and Codegen outperform previously developed deep learning networks across all studied languages. Furthermore, we contribute a new manually curated benchmark set of 100 examples of inconsistent comments from real-world pull requests and perform a field study by submitting 20 pull requests that fix detected issues. This benchmark set was constructed by analyzing a newly mined dataset of more than 13 million comments from GitHub pull requests. On this hand-crafted benchmark evaluation set, we observe lower performance from all models than observed during training, which highlights the need to consider real-world conditions when building models. To study how developers value fixing the problem of inconsistent comments, we sent 20 pull requests to open source projects on GitHub. With 90% of pull requests accepted, we find that developers are receptive to automated comment consistency detection, validating its purpose in the open source setting.

The code and data used in this project are published on GitHub.[1]

[1] `https://github.com/pelmers/llms-for-code-comment-consistency`

Preface

This thesis was written for the graduation project for the Master's in Data Science and Artificial Intelligence study at Eindhoven University of Technology (TU/e). This work represents the culmination of a two year program, and I could not have made it here alone. I would like to warmly thank my thesis advisors, my friends, and my family who supported me on this journey.

First and foremost, my deepest gratitude goes to my advisor Ir. Zeno van Cauter. He committed a tremendous amount of time and effort to helping me with thesis revision. His meticulous attention to detail helped shape every aspect of this work, from the beginning to the end.

Equally, I also want to thank my advisor Prof. Alexander Serebrenik. His profound knowledge of the field of software engineering meant that for any problem I encountered, he knew a path to the answer. There was never an instance where he did not point me directly to the most relevant previous work for any question.

Certainly not least, I would also like to extend my appreciation for my advisor Prof. Mykola Pechinizkiy. In addition to giving me valuable feedback throughout the project, he should also take credit for leading a research group that fosters learning and growth.

I also cannot forget to express gratitude towards all of the friends I spoke to during the course of this thesis project, who helped me sound out ideas and methods. Finally, I thank my mother and sister for supporting my choice to travel internationally to the Netherlands to pursue this study.

Peter Elmers
Eindhoven,
September 2023

Contents

Contents vii

List of Figures ix

List of Tables xi

Listings xiii

1 Introduction 1
- 1.1 Context . . . 1
  - 1.1.1 Code and Comments . . . 1
- 1.2 Motivation . . . 2
  - 1.2.1 Programming Bottleneck . . . 2
  - 1.2.2 Multiple Programming Languages . . . 2
  - 1.2.3 Application of Large Language Models . . . 2
- 1.3 Thesis Organization . . . 3

2 Problem Formulation 5
- 2.1 Problem Statement . . . 5
- 2.2 Problem Scope . . . 7
  - 2.2.1 Language Selection . . . 7
  - 2.2.2 Real-world Context . . . 8
- 2.3 Limitations of Past Work . . . 8
  - 2.3.1 Focus on Java . . . 8
  - 2.3.2 Lack of deep learning-based approaches . . . 8
  - 2.3.3 Practical Applications . . . 9
  - 2.3.4 Developer Responses . . . 9
- 2.4 Research Questions . . . 9

3 Background 11
- 3.1 Comment Consistency Analysis . . . 11
  - 3.1.1 Definitions of Code Comment Consistency . . . 11
  - 3.1.2 Measurements of Comment Quality . . . 12
  - 3.1.3 Comment Generation . . . 13
- 3.2 Code Language Models . . . 13
  - 3.2.1 Language Models . . . 13
  - 3.2.2 Architecture Introduction . . . 14
  - 3.2.3 Code-Specific Training . . . 14
  - 3.2.4 Fine Tuning . . . 15
- 3.3 Software Data Collection . . . 15
  - 3.3.1 GitHub Mining Practices . . . 15
  - 3.3.2 Empirical Approaches to Code Evaluation . . . 16
- 3.4 Evaluating Contributions in a Social Setting . . . 17

4 Methodology **19**
4.1 RQ1: Model Training and Synthetic Evaluation . . . 19
4.1.1 Comment Selection . . . 19
4.1.2 Dataset . . . 20
4.1.3 Model Selection . . . 21
4.1.4 Evaluation . . . 23
4.2 RQ2: Multilingual Data Collection . . . 25
4.2.1 Dataset Collection . . . 25
4.2.2 Modelling Decision . . . 27
4.2.3 Evaluation . . . 27
4.3 RQ3: Practical Benchmark Creation . . . 28
4.3.1 Pull Request Comment Mining . . . 28
4.4 RQ4: Real-world Performance . . . 29
4.5 Ethical Considerations . . . 30

5 Results **31**
5.1 RQ1: Tests on Existing Java Dataset . . . 31
5.1.1 Analysis . . . 32
5.2 RQ2: Extension to Other Languages . . . 33
5.2.1 Dataset Details . . . 33
5.2.2 Necessity of newly trained models . . . 36
5.2.3 Model Results . . . 36
5.2.4 Model Analysis . . . 36
5.3 RQ3: Benchmark Evaluation . . . 38
5.3.1 Overview . . . 38
5.3.2 Pull Request Comment Details . . . 38
5.3.3 Results . . . 39
5.4 RQ4: Social Study Results . . . 39
5.4.1 Pull Request Analysis . . . 40

6 Discussion **41**
6.1 Newly Collected (**RQ2**) Dataset Details . . . 41
6.1.1 Dataset Validation . . . 41
6.2 Model Discussion . . . 42
6.2.1 Replication . . . 42
6.2.2 Explanations of Low Performance . . . 42
6.2.3 Negative Results in Context . . . 43
6.3 Benchmark Findings . . . 44
6.3.1 Validity of Results . . . 44
6.3.2 Generality . . . 44
6.3.3 Challenges . . . 45
6.4 RQ4: Social Study . . . 45

7 Conclusions **47**
7.1 Future Work . . . 47

Bibliography **49**

Appendix **53**
A.1 Data Filtering Examples . . . 53
A.2 Submitted Pull Requests . . . 55

List of Figures

2.1 Code Comment Consistency Detection 5
2.2 Example of reviewed pull request on GitHub 7
2.3 Number of software repositories by language on GitHub[6] 7
2.4 Word clouds of topics by language 9

3.1 Model architecture of Panthaplackel et al.[28] 12
3.2 BERT input representation[9] 14
3.3 Example problem from MTPB[26] 16

4.1 Example of a commit that modifies both the code and the comment 21
4.2 CodeBERT model diagram 22
4.3 CodeGen model diagram 23
4.4 Example of saliency map visualization from sentiment analysis [8] 24
4.5 Example pull request associated with **RQ4** 30

5.1 Histograms of length distributions in training and validation sets of replication data . . . 31
5.2 CodeBERT Training Loss and Validation F1, each epoch measured in 100 steps 32
5.3 Example Gradient Norm Visualization, trailing newlines highlighted 33
5.4 Example Gradient Norm Visualization, trailing newlines removed 33
5.5 Histograms of length distributions in training sets by language 35
5.6 CodeBERT Training Loss and Validation F1 37
5.7 Example Comment Fix Pull Request 40

List of Tables

3.1 Key Findings from Kalliamvakou et al.[16] (2014) 16

4.1 Balanced Confusion Matrix of 100 samples 24
4.2 Confusion Matrix with 100 samples, 20:1 negative:positive ratio 24
4.3 CodeBERT trained on all 4 languages or only Python, evaluated on Python test set 27

5.1 **RQ1** Test Performance Comparison 32
5.2 Dataset Statistics 34
5.3 Mean and Median Dataset Lengths (number of tokens under CodeBERT tokenization) . 34
5.4 New Line Results in Java 35
5.5 CodeBERT trained on replication data, tested on new data 36
5.6 CodeBERT trained on whitespace-normalized replication data, tested on new data 36
5.7 Weighted F1 Scores by Model and Programming Language 37
5.8 Benchmark Statistics 38
5.9 Benchmark Set Weighted F1 Scores, compared with test set overall score (from Table 5.7) 39
5.10 Pull Request Outcomes 40

Listings

1.1 Documentation comment example in Python 1
1.2 Inline comment example in Python 1
2.1 Consistent Documentation comment example 6
2.2 Inconsistent Documentation comment example 6
4.1 Comment types 20
4.2 Before change example 26
4.3 After change example 26
4.4 Before change example 26
4.5 After change example 26
5.1 Python Example (Before) 34
5.2 Python Example (After) 34
5.3 Java One-Line Example (Before) 34
5.4 Java One-Line Example (After) 34
5.5 Example from Go benchmark 38
6.1 Java Consistent Comment Example (Before) 41
6.2 Java Consistent Comment Example (After) 41
6.3 Python Example (Before) 43
6.4 Python Example (After) 43
6.5 Go Example (Before) 43
6.6 Go Example (After) 43
6.7 Java Benchmark Example (Before) 44
6.8 Java Benchmark Example (After) 44
A.1 `Generated Code` 53
A.2 `Generated Code` 53
A.3 `Duplicated Code` 54
A.4 `Code with self-admitted technical debt` 54
A.5 `Deprecated Code` 54

Chapter 1

Introduction

In this section, we will establish a context for the study of code comment consistency and provide some motivation for the project.

1.1 Context

1.1.1 Code and Comments

In software development, comments are text annotations embedded into code for the benefit of humans. Whereas code itself is interpreted by a machine into executable instructions, comments in source code are ignored by the machine and in most cases only provide value for other humans. Because these comments are ignored by the computer and thus do not need to follow any structure or syntactic rules, they can vary greatly in purpose and style.

```
def gcd(a, b):
    """Find the greatest common divisor of integers (a,b).
    """
    m = min(a, b)
    for i in range(m, 0, -1):
        if a % i == 0 and b % i == 0:
            return i
```

Listing 1.1: Documentation comment example in Python

Listing 1.1 shows an example of a comment in the Python language. This type of comment is known as a *documentation comment*, which in general will describe the overall functionality of a function without going into very specific details about its implementation. Listing 1.2, also written in Python, introduces the *inline comment*. This type of comment is usually written to give more detailed documentation about specific lines of code, for example to explain the reasons for a particular choice or to give a more easily understandable alternative to a particularly complex section. In this example, the math formula involving subtraction and the modulus operator is explained in words.

Chapter 2 gives a more precise definition of the particular type of comments considered in this study.

```
def pad_with_zeros(str_num):
    '''
    Return a string with zeros inserted at the start of the string str_num
    such that its length is divisible by 3.
    '''
    if len(str_num) % 3 == 0:
        return str_num
    else:
        # Insert either 1 or 2 '0' characters by taking the remainder of division by 3
        # and turning 1 into 2, 2 into 1.
        return '0'*(3 - len(str_num) % 3) + str_num
```

Listing 1.2: Inline comment example in Python

1.2 Motivation

1.2.1 Programming Bottleneck

The key insight which underlies the motivation of this project is that in real industry applications, developers spend much more time reading code than writing it. In fact, up to 83% of a professional developer's programming time is spent on code navigation and understanding.[48] In a company, every line of code is written once but must be understood by every developer who will use it. To accelerate the process of understanding, disciplined programmers will add comments to their code. Comments in code are natural language statements which are ignored by the program execution and are intended solely for the benefit of the reader. A well-written and up-to-date comment can therefore let the reader understand the intention of the adjacent code without needing to fully trace out each line the computer will run.[37]

On the other hand, poorly written, mistaken, or obsolete comments can lead to immense frustration and accumulation of technical debt.[7] When other developers read such comments, they would assume the behavior matches the written description, but then encounter surprise when the program performs something else. In some cases they may not even notice at first, leading to even greater debugging difficulty in the future as more and more layers of code builds upon it before the mistake is eventually uncovered.

1.2.2 Multiple Programming Languages

In the modern software industry, companies choose to use programming languages for their characteristic strengths and weaknesses.[27] This specialization of tooling means that even within a single company, multiple programming languages may be used for different purposes. However, existing research in the field of software comment analysis focuses heavily on the Java language[30], where the specific style of commenting and development patterns may not carry over to other languages. For example, method comments in Java are often partially generated by integrated development environments (IDEs), which give a uniform style and consistent standards even across different projects. Java also tends to have longer variable names than other languages which have a stronger focus on succinctness.

Novelty is another motivation for the topic of this study. As far as we could identify, no accessible and labeled dataset of code and comment pairs exists for languages outside of Java.[49] The creation of such datasets could encourage future work in the topic of studying code comment consistency across a broader range of language contexts.

1.2.3 Application of Large Language Models

Large language models are mathematical representations of language that are capable of making textual predictions such as the next word in a sentence or whether a given text review is positive or negative. The *large* in the name refers to the number of numerical parameters in the representation, normally at least several hundred million values. For example, ChatGPT is a large language model. Given recent successes in applying large language models[24] in text analysis settings, we see potential in harnessing them for the task of detecting inconsistent comments, thus improving the quality of software and the development experience. Most past research in source code comment classification has used heuristic techniques based on factors such as the number of overlapping words between the comment and the function code, leaving room for exploration of machine learning methods.[30] Although using a large language model is not a guaranteed method for success by any means, we are motivated by the technique's success in other contexts and see its potential in this problem setting as well.

1.3 Thesis Organization

The rest of the thesis is organized as follows.

- Chapter 2 formulates the main research goals and poses the core research questions for this project.
- Chapter 3 provides background information from academic literature of current work on the topic.
- Chapter 4 describes the methodology followed in this project to answer the primary research questions.
- Chapter 5 presents the results of work on each research question with analysis.
- Chapter 6 discusses the validity of the found results, investigating potential explanations and highlighting possible risks.
- Chapter 7 concludes the work by summarizing the methods and findings of the paper.

Chapter 2

Problem Formulation

In this chapter we formulate the problem and present our research questions. In Figure 2.1 we illustrate the overarching goal of comment consistency detection in terms of its inputs and outputs.

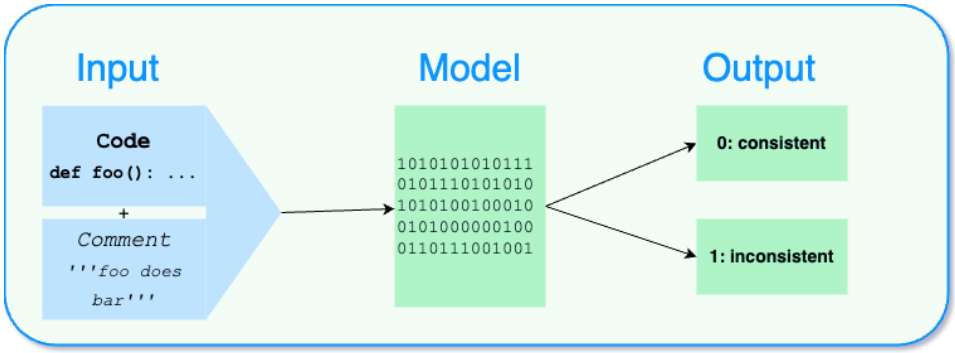


Figure 2.1: Code Comment Consistency Detection

2.1 Problem Statement

A critical component of effective software development is documentation of source code. The most time-consuming activity of a professional developer is code navigation and understanding, and comments in code directly help with this task.[48]. As software projects grow, new code is written and old code is modified. However, whereas the execution of the software tests the function of the code, there is no mechanism to ensure that comments remain consistent with the code they are describing.

The problem this research project seeks to address is how to apply large language models to the *real-world* problem of detecting *inconsistent* comments across code in multiple programming languages.

Meaning of Consistency In this description, the word *inconsistent* deserves a section to clarify its meaning.

Definition 1. *A comment is* consistent *with its associated code if a qualified expert reviewer explicitly deems it as such.*

This definition encompasses a broad spectrum of potential sources of inconsistencies. A qualified expert reviewer should be someone who is not the author of a code change but is also knowledgeable in the project in which the change was authored. The concept of consistency proposed in this formulation most closely matches the definition of *coherence* introduced by Steidl et al.[37] While every reviewer

will have his or her own standards, we expect that this definition will align several commonly studied assessment metrics, including understandability, completeness, and accuracy. [30]

We demonstrate an example of the difference between a consistent and inconsistent comment in Listing 2.2. In this example, we reproduced the correct example of Listing 1.1 along with a version with a modified comment to give an example of both a consistent and an inconsistent comment. The comment is inconsistent because the code returns the greatest common divisor, but the comment refers to the least common multiple.

```
def gcd(a, b):
    """Find the greatest common divisor
       of integers (a,b).
    """
    m = min(a, b)
    for i in range(m, 0, -1):
        if a % i == 0 and b % i == 0:
            return i
```

Listing 2.1: Consistent Documentation comment example

```
def gcd(a, b):
    """Find the least common multiple of
       integers (a,b).
    """
    m = min(a, b)
    for i in range(m, 0, -1):
        if a % i == 0 and b % i == 0:
            return i
```

Listing 2.2: Inconsistent Documentation comment example

However, although Definition 1 gives the ideal method of determining whether a comment is consistent, is not practical in a supervised learning setting. Because it would require asking an expert for their opinion for every potential example, we could not build an adequately large dataset with which to train a model. Therefore, we introduce the following definition, given by Panthaplackel et al.[28], which is more conducive to a weak supervision setting.

Definition 2. *Given a version control change that modifies both a function's code and its documentation comment, we define the old comment as inconsistent with the new code, and the new code as* consistent *with the new comment.*

In effect, one of the outcomes of this project is a measurement of the alignment between these two definitions. This work follows the model of previous studies by assuming Definition 2 can reasonably substitute for Definition 1. In this work, we thus refer to Definition 1 as the gold standard and Definition 2 as the silver standard.

Meaning of Real-world Another vital term to clarify from the problem statement is *real-world*. Specifically, the term *real-world* in this case refers to matching an expected use case of software development as closely as possible. For this project, the use case of choice is code review. Code review is a standard practice in the software industry where someone other than the author of a piece of code reviews it and decides whether to approve it or not. In Figure 2.2, we present an example of a pull request that was accepted in review. In this case, *r+* is a shorthand expression meaning approval.

At this stage, one of the aspects considered is the quality of comments. In practice, the goal of this project is to create an additional layer of code review that is specialized for checking comments. This is important because even after code review, researchers have observed examples of code commits which solely fix mistaken comments.[44] This phenomenon indicates that a tool which can reliably discern consistent from inconsistent comments at review time would improve developer productivity by reducing the time spent later fixing previous mistakes.

Thus, a *real-world* evaluation of comment consistency checking would involve testing it against examples of code review scenarios.[49] Chapter 4 expands upon the exact procedures for finding and extracting these examples from software projects.

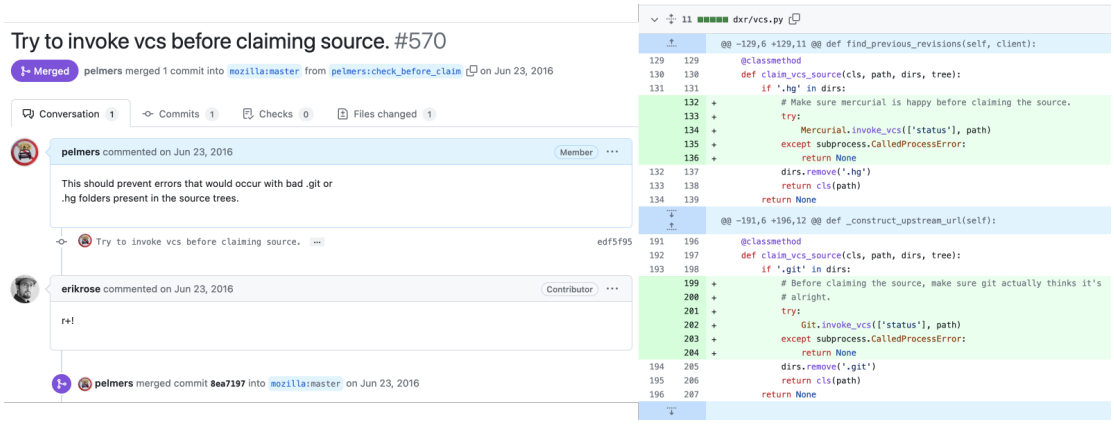


Figure 2.2: Example of reviewed pull request on GitHub

2.2 Problem Scope

This section explains the project scope in terms of the studied languages, and the setting of the real-world aspect of this research.

2.2.1 Language Selection

Every programming language has its own syntax and conventions. A model trained on Python code will likely not perform as well when it receives an input of JavaScript code. Java comments, for example, follow the well-defined Javadoc standard [38] and appear before the function expression, while Python function comments are formatted differently, written as a string after the definition line.

With the focus of previous work on the Java language, one goal of this project is to generalise the model to *Python, JavaScript*, and *Go*. This language selection is motivated by popularity in the open source community. Selecting languages by popularity promises the advantages of the broadest applicability and the most availability of training data. As a preliminary for this project, we mined the metadata for more than 3 million GitHub repositories[6], consisting of all public repositories with at least 5 stars on GitHub from 2009 until May 2023. Figure 2.3 plots a bar chart of the number of repositories by language on GitHub.

Note that TypeScript is a syntactically very similar superset of JavaScript, so for the purposes of this analysis we will consider the two languages together. Also, we selected Go instead of PHP or C,

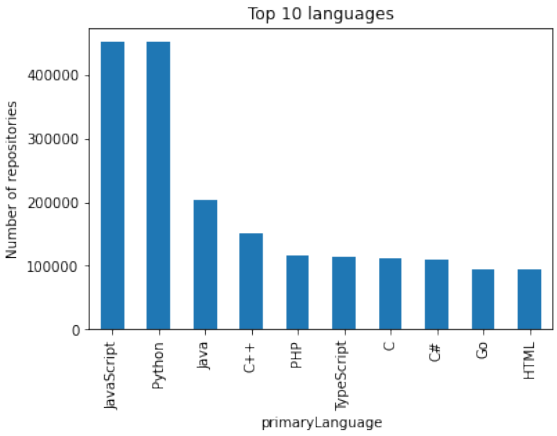


Figure 2.3: Number of software repositories by language on GitHub[6]

which ranked ahead of it on strict popularity, because Go uses a standardized documentation format with *godoc*. [14]

2.2.2 Real-world Context

The second half of the problem statement pertains to the real-world aspect of the research, where we define the real-world use case as open source development on GitHub. This scope includes two distinct contexts.

- Evaluation of existing examples: existing open source development contains a rich collection of expert annotated comment consistency examples in the form of past pull request review. In other words, we can retrieve previous examples of fixed inconsistent comments as a result of expert review and test whether our methods produce a tool that would also correctly identify the same mistake.
- Human response to novel contributions: a new developer tool is only useful if human developers are willing to use it. Therefore, part of answering the problem statement is to find out how people react to the usage of such a tool. In this case, we model usage of the tool in the real world in the open source world via pull requests.

By considering both existing and new examples in the open source context of GitHub pull requests, this research aims to assess the ability of the proposed approach to detect inconsistent comments in the real world.

2.3 Limitations of Past Work

2.3.1 Focus on Java

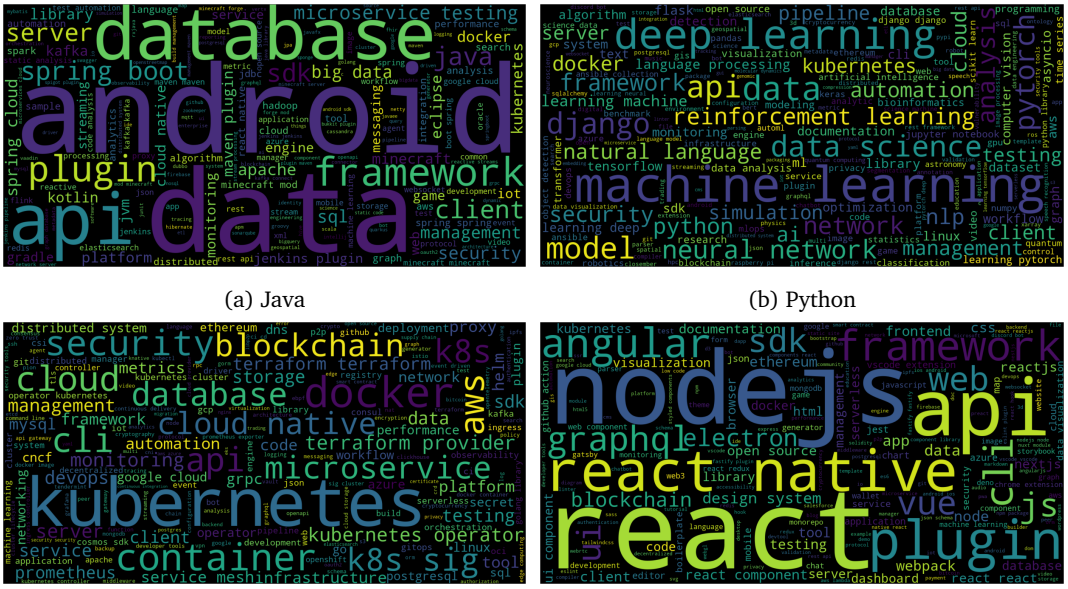
A systematic literature review by Rani et al.[30] surveyed the research in this field from 2011 to 2020. From the 47 studied papers, the authors identified several trends and gaps where future work could focus. For example, 87% of existing work studied the Java programming language. Indeed, all previous works that relate most closely to the setting in this project were studies conducted on the Java language. In a polyglot software ecosystem this singular focus on one language might lack important context in other languages or lead to assessment methods that only work in Java.

Figure 2.4 visualizes the most popular project categories per language with word clouds of each project's self-selected topic field on GitHub, generated from analysis of the metadata of over 3 million repositories.[6] In addition, each language has its own niche in the open source ecosystem with very little overlap. That means studies conducted on one language where most projects are focused on one domain may not generalize to another language with a different market. For example, we can see that Java has a heavy focus on Android and databases. Python is oriented towards machine learning. Blockchains and cloud services are very popular in Go, and JavaScript is dominated by React and Node.js.

Because previous works have only collected datasets for Java, study of other languages will require collection of new datasets through software mining methods.

2.3.2 Lack of deep learning-based approaches

Past work in the study of code comments has most often used the technique of manual assessment to evaluate quality.[30] While the fully manual approach has the advantages of clear interpretation and high accuracy, this work requires considerable effort to produce results. This high workload has contributed to a lack of existing tools that can aid developers in writing consistent comments and code. In other fields related to natural language processing, deep learning based approaches have led to results that surpass previous methods, such as in news categorization and sentiment analysis.[24] Most recently, researchers have begun to apply language models to the task of comment consistency



(a) Java (b) Python

(c) Go (d) JavaScript

Figure 2.4: Word clouds of topics by language

evaluation.[28][39] Compared to the extensive history of traditional methods to evaluate comments, the use of deep learning techniques for this task has a large space left to explore. To address our problem statement, we extend beyond previous approaches by incorporating the use of large language models that have been specifically trained on code.

2.3.3 Practical Applications

The research survey conducted by Rani et al. also identified a deficiency in evaluative work as a gap in the current field of study.[30] Past works which have proposed comment quality metrics have not evaluated them in a representative setting, or have only evaluated them on available computer science university students.[7] In this study, the main aim of studying **RQ3** is to fill this gap in the current literature in addition to addressing the *real-world* component of the problem statement. Our goals are to answer how well current methods perform on representative examples and how practitioners respond to proposed comment consistency contributions.

2.3.4 Developer Responses

Related to the previous point on practical applications, we find that previous works have not deeply explored the responses of developers to comment consistency checking systems. Although we can evaluate metrics such as accuracy and precision with examples from real-world projects, these figures would not illuminate whether developers truly find the contributions useful. Therefore, we include **RQ4** to investigate reactions of developers to contributed fixes to comment consistency in a code review setting.

2.4 Research Questions

The remainder of this thesis seeks to answer the following research questions. Each research question is specifically targeted at an identified gap in previous work.

- **RQ1.** How can we apply large language models (LLMs) to classify comment consistency?

- **RQ2.** How can we evaluate consistency on languages outside of Java?
- **RQ3.** How do we create a benchmark that matches real-world situations?
- **RQ4.** How do developers react to the results of this model?

Chapter 3

Background

This chapter provides an in-depth look at background literature around the topics of code comment analysis and applied language models. In doing so, we will also highlight gaps in existing work to motivate the novel contributions of this project.

3.1 Comment Consistency Analysis

In this section we provide an overview of previous work on the study of code comment consistency.

3.1.1 Definitions of Code Comment Consistency

The unlimited flexibility of comments creates a massive surface area for the study of code comment quality. Past work has focused on narrowing down the scope of the problem to specific programming languages or types of comments. For example, while inline comments often describe something about the behavior of code, they may also note behavior that has not been implemented yet. Some languages such as Java impose a structure to certain comment types, such as Javadoc-style method level comments. Comment analysis could use knowledge of this structure to assess comment quality, for example by checking that the `@param` variable name matches the value present in the code.

The field of evaluating comment quality is inherently subjective and many potential definitions of consistency are available.[30] In addition to consistency, studies have focused on overlapping qualities including relevance, accuracy, completeness and accuracy. The fuzziness of terminology and overlap between studied aspects of code and comments has historically made it difficult to compare results from different works in the field of code comment consistency.[30] For illustration, we share a selection of examples of closely related quality definitions below, as noted by Zhi et al.[52] The purpose of this short list is to indicate that the study of code comment consistency has not settled on a single definition for what constitutes a good or bad comment. Instead, it is still highly subjective.

- **Correctness**: Whether the information in the comment is correct.[52]
- **Accuracy**: Accuracy or preciseness of the comment. For example, it should not be too abstract or vague.[52]
- **Relevance**: How relevant the comment is to a given purpose.[30]
- **Usefulness**: The extent to which the comment can be used by its readers to achieve an objective. [37]

In this project, we defined consistency in Definition 1 and Definition 2. This definition is shared by several past works in the field and thus allows direct comparison of results, including Liu et al.[20] and Panthaplackel et al[28]. The necessity of gathering large amounts of data to train large models means that we cannot feasibly look at every single example of code and comment to decide whether they are

consistent or not. Definition 2, along with selective preprocessing and careful consideration of which open source project to include, provides a heuristic to determine whether a comment is consistent.

3.1.2 Measurements of Comment Quality

In the previous section we discussed background in the definitions of consistency. We shared examples of how different past works have defined the quality rules they studied. In this section we survey measurements of comment quality. Note that the methods of measurement may not align exactly with the definitions of quality. Normally, the definition of quality is a subjective statement (*the comment is useful*), but the measurement is objective (*the comment has 0 mispelled words and 1 formatting mistake*).

Heuristic-based approaches Rani et al. highlighted the importance of defining relevant metrics for assessment.[30] Previous works have used a variety of objective metrics to measure subjective comment quality. For example, to measure the quality of *understandability*, the Flesch-Kincaid readability score has been used.[32] In another paper, Scalabrino et al. used word overlap proportion between code and comment as a measure for the subjective quality of *relevance*.[33]

However these metrics may not generalise to practical usage as an indicator for quality.[30] A large overlap between code and comment might imply the comment is redundant. The readability score, which is determined from the lengths of words and sentences, may be meaningless in case the comment contains code as well.

Deep Learning Approaches In the closest comparison to the work outlined in this thesis, Panthaplackel et al. applied deep learning models to this problem of comment consistency.[28] In deep learning, measured quality of a comment no longer corresponds to an easily explainable rule. The validity of such a measurement depends on how well it performs and the applicability of the circumstances under which the model was trained.

In their work, the problem of consistency of JavaDoc method level comments was studied by constructing a balanced dataset of 40,688 examples sampled from 1,518 open-source Java projects. The authors built a set of transformer-based models that encoded the code text with either a sequential text model or a graph neural network (GNN) that incorporated AST information. An overview of this architecture is depicted in Figure 3.1.

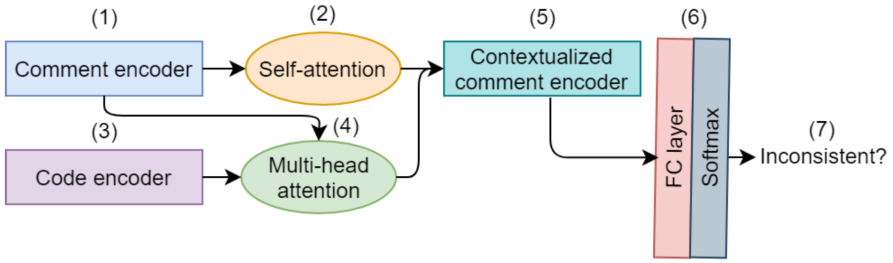


Figure 3.1: Model architecture of Panthaplackel et al.[28]

The authors evaluated their methods in two circumstances, defined as *post-hoc* and *just-in-time*. In the post-hoc setting, the model only receives the final comment and code pair and must determine whether they are consistent. This setting matches the problem definition of our work. In the just-in-time setting, the input to the model also includes the previous value of the comment and code. This setting corresponds to the use case of detecting inconsistencies at commit time or while editing. On the full test dataset, the best model (GNN-based) achieved an F1 score of 0.672 in the post-hoc setting and 0.809 in the just-in-time setting. However, these results include evaluation on the *structured* `@param` and `@return` sections of the Javadoc format. As discussed in Chapter 2, our setting only studies the unstructured portion of the comment which cannot be very easily analyzed with standard development

tooling. Therefore, when considering only the summary line of the JavaDoc comment, the best model had an F1 score of 0.706. The authors reported that this model took 30 minutes to train on an Nvidia Titan GPU.

The next piece of related work we consider is an improvement of these results using BERT and Longformer. Using the same dataset and evaluation metrics as Panthaplackel et al., Steiner et al.[39] fine tuned the pre-trained base BERT model on this task. In the post-hoc setting, the authors achieved a best F1 score of 0.72. Training this model took place in a distributed fashion across 8 Nvidia RTX A5000 GPUs (24 GB each) for an unspecified amount of time.

In addition to building new models based on LLMs, our work replicates the models created by Panthaplackel et al. and Steiner et al. for comparison on new datasets.

3.1.3 Comment Generation

Comment generation refers to automatically generating comments for existing code that would help programmers understand, reuse, and maintain software. While our project does not specifically target the outcome of generating comments, it is still relevant to understand recent background in this task. That is because one could frame comment consistency as a problem of generation by stating a comment is consistent if an automatically generated comment is similar enough to the original.

In a recent survey of comment generation technology, Song et al. [35] identified several approaches which are similar to the methods seen for comment consistency detection. For example, code clone techniques look for semantically similar code across a code base and use existing comments to fill gaps in missing functions. Deep neural network approaches learn to generate comments by training on datasets of code and comment pairs. This field faces similar challenges in the subjectivity of evaluation criteria and difficulty of measuring practical impact.[35] While work on comment generation continues, a recent survey by Stapleton et al. demonstrated that human written comments were more helpful than machine generated comments at program comprehension tasks.[36]

In our project, we choose to focus on comment consistency detection rather than generation because of the overarching goal as a tool to enhance code review, where code is already written and must be checked for correctness. Another reason we include a short background on comment generation is because the datasets used to study one problem can largely be used to study the other as well. In both cases, datasets are composed of comment and code pairs. For example, the datasets created in this work could also be used to train comment generation models for languages outside of Java.

3.2 Code Language Models

Because computer code is simply text, language models are an ideal candidate for performing tasks based on source code. In this usage, a large language model (LLM) refers to a deep neural network trained on a task involving natural language understanding and at least several hundred million parameters.

3.2.1 Language Models

In general, language models are statistical representations of language relationships. While numerical language models such as N-gram Markov chains have been used for many decades for text generation [21], recent computing advancements have fueled an incredible proliferation of large scale transformers-based textual models trained on vast quantities of Internet-sourced datasets, such as BERT[5] and the GPT family.[29]

Collectively, although the exact meaning of *large* is fuzzy, these new models are known as large language models because they are composed of millions or billions of numerical parameters. These models learn language relationships by turning words into numbers (a step known as encoding) and optimizing their ability to predict tokens from a large corpus of text through gradient descent. Because

text is also the representation of programming language source code, a natural consequence is to employ these models to program code. Approaches such as OpenAI Codex[3] now already see widespread usage as the basis for the Copilot autocompletion tool.

3.2.2 Architecture Introduction

Two prominent examples of large language models which feature in this project are BERT and GPT. Both are based on a transformer architecture and were trained to predict tokens in large corpora of internet-sourced text (billions of words). One key advancement that enables the scale of these models is that they are self-supervsed, meaning that a human does not need to label the correct prediction during training. Instead, they are evaluated against predicting tokens that already exist in the dataset's text but are hidden from the model's input.

Training Details This project applies one model based on BERT and one model based on CodeGen to study the problem of comment consistency. While both models are based on transformers and model input dependencies through the attention mechanism[43], BERT is an encoder and CodeGen (like GPT) is a decoder. That means the output of BERT is one embedding vector per input token, thus input and output have the same length. In contrast, the output of CodeGen is a single vector representing a probability distribution for a prediction of the next word following the given input.

First, we will convey some of the relevant training and implementation details of BERT (which CodeBERT is based on). The first input token to BERT is the special class token, `[CLS]`, followed by the input sequence tokenized with WordPiece and a vocabulary size of 30.000. This representation is shown in Figure 3.2. The model is trained with the masked language modeling (MLM) task on a dataset consisting of BookCorpus and English Wikipedia. In this task, 15% of input tokens are hidden with a special `[MASK]` token, and the goal of the model is to predict the hidden tokens. Note that during this stage a single projection layer which produces a vocabulary-sized probability distribution is added to the model.

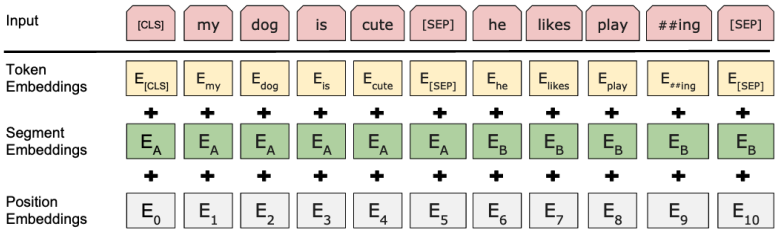


Figure 3.2: BERT input representation[9]

3.2.3 Code-Specific Training

While the datasets used to train these models did contain code samples, for specialised use in code-related tasks additional training gives better performance. Thus, CodeBERT[9] and Codex[3] emerged from BERT and GPT, respectively, by fine tuning on programming language datasets.

CodeBERT CodeBERT applies the same overall architecture as BERT base, with 110 million trainable parameters. However, it uses a byte-pair encoding as its tokenizer instead of WordPiece. Most relevantly to the work in this project, CodeBERT also uses a dataset sourced from publicly available GitHub repositories in six programming languages: Java, Python, Go, JavaScript, PHP, and Ruby. Its pre-training task is *bimodal* MLM, where a natural language and related code input are joined with a special `[SEP]` token and fed together to the model. These inputs are sourced from functions and their documentation comments. Indeed, the pre-training step in CodeBERT nearly identically resembles the

input dataset of our project. Thus, we may expect that it can be very efficiently adapted to the task of inconsistency detection.

CodeGen The second model we will investigate, CodeGen[26], is a publicly available cousin of Codex, which is not accessible for fine tuning because of an exclusive agreement between OpenAI and Microsoft. The CodeGen project publishes several model sizes, of which we will only plan to use the smallest two for evaluation purposes (350 million and 2 billion parameters). The input to this model is a byte-pair encoding tokenized string, and its output is a probability distribution over the vocabulary for the next token. In the training stage, the objective of the model is to predict the next token of each input sample. This multilingual model is sequentially trained over two datasets. First, it is trained on ThePile, a 354 billion token dataset consisting of an agglomeration of many smaller language datasets.[11] Then, it is trained on BigQuery, a dataset consisting of open source C, C++, Java, JavaScript, Python, and Go code.

Note that the intersection of common languages between CodeGen and CodeBERT are exactly the languages we propose to study in this project.

3.2.4 Fine Tuning

For applications other than token prediction, we apply fine-tuning with labeled datasets against the pre-trained model. This process produces a model that performs a *downstream task*, such as question answering, sentiment analysis, or, in the case of this project, code-comment consistency detection. The intuition behind why this procedure can work is that through pre-training, the model has already learned how to make appropriate connections (weights) between its inputs that gives it an understanding of the vocabulary and grammar of the underlying language. Thus the model only needs a relatively small amount of additional supervision to be applied in a related downstream setting. Evaluations of these fine-tuned models in similar tasks to comment detection such as code search have shown similar performance to specialized approaches designed specifically for that task.[9]

3.3 Software Data Collection

This section covers literature relevant to the topic of mining code from open source repositories. Though not the main focus of this project, understanding past work on collecting software data for research is a critical dependency due to the lack of existing data for comment consistency detection across multiple languages.

3.3.1 GitHub Mining Practices

GitHub is the de-facto online repository for a vast amount of software source code, and it is free to use for open-source projects. However, with this accessibility comes a set of challenges to the researcher studying the practice of software engineering. In *The promises and perils of mining GitHub*, Kalliamvakou et al. present a set of considerations for selecting which open source software projects to study.[16]

Since the goal of this project is to produce a model that can apply to professional applications, care must be taken to filter down the set of considered repositories. Simply picking the most-starred projects may not lead to a representative dataset. For example, the most-starred project on GitHub, `freeCodeCamp/freeCodeCamp`, is an online programming course. But this project's seemingly overwhelming popularity is manufactured by the curriculum itself, where one of the first tasks is to star this repository on GitHub.

To perform the data collection component of this project, we also need to understand relevant software engineering research techniques. In *Promises and Perils of Mining GitHub*, Kalliamvakou et al.[16] map out many features to be aware of when mining data from GitHub. In the paper, the authors explain that the nature of GitHub as a free platform for any open-source software leads to

many repositories that are inactive, personal in nature, or not even containing software at all. Some of the key findings are highlighted in Table 3.1.

Peril Category	Percentage (%)
Inactive projects	46
Non-software development	36
Personal projects	72
Less than 25 total pull requests	95

Table 3.1: Key Findings from Kalliamvakou et al.[16] (2014)

3.3.2 Empirical Approaches to Code Evaluation

The goal of creating a benchmark set also deserves some attention in background research. In a large empirical study of 500 commits, Wen et al. created a taxonomy of changes and identified which types of commits were most likely to introduce code-comment inconsistencies.[44] Specifically, the authors mined examples of potential issues by looking for commit messages containing the key words *update* or *outdate* and *comment*. This method led to a false positive rate of 27.6% among the 500 manually analyzed commits, where the pattern suggested a relevant comment but inspection revealed it was unrelated.

Along with the Codex model, Chen et al. [3] introduced the HumanEval benchmark, a collection of 164 original programming problems in the style of software developer interview questions. In this benchmark, models are evaluated based on their abaility to produce code with the same output as the human created solution when executed.

In addition to their CodeGen model, Nijkamp et al. built a benchmark known as the Multi-Turn Programming Benchmark (MTPB).[26] This benchmark is similar to HumanEval in that it also evaluates models by comparing the output of their generated code given a problem statement. However, these problems are given in a turn-based fashion, where prompts are given for several steps of the overall problem. This benchmark contains 115 problems. Figure 3.3 shows an example of one problem from MTPB.

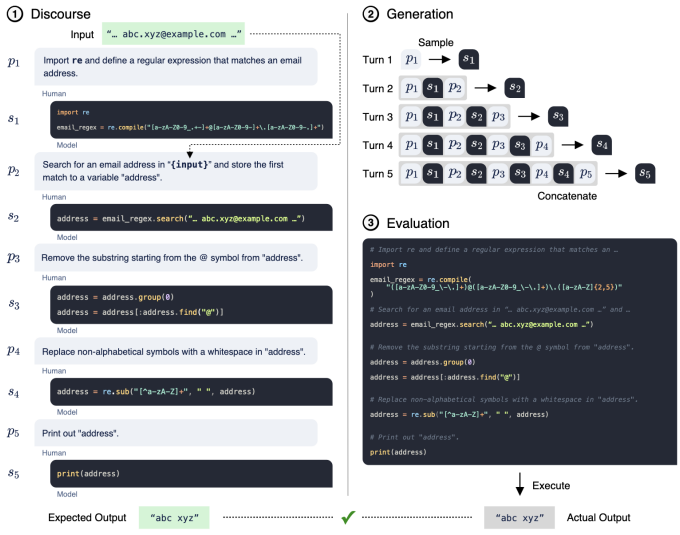


Figure 3.3: Example problem from MTPB[26]

3.4 Evaluating Contributions in a Social Setting

One expected outcome of the research in this project is an understanding of how developers react to contributions that fix comment inconsistencies. While this specific type of contribution has not been studied in the past, we can look at other works that have submitted fixes to open source projects on GitHub. In the context of our work, the purpose of reviewing previous research on pull request contributions is to inform expectations about likely acceptance rates and to estimate an adequate sample size.

Marcilio et al. developed a bug fixing tool based on warnings emitted by popular code static analysis tools such as FindBugs and SonarQube. [22] The authors submitted 38 pull requests to open source Java projects, including the Eclipse IDE and FindBugs tools, and found that 84% of suggested fixes were accepted and merged. In a similar work, Rolim et al. mined project revision histories on GitHub to create `REVISAR`, a tool that identifies frequently used edit patterns. The authors submitted 16 pull requests using the tool, of which 6 pull requests were accepted. Tan et al. piloted an innovative approach by integrating the process of fixing open-source bugs into an academic curriculum. [15] By sourcing contributions from a large class of 154 students, the authors achieved a greater scale of pull requests than other works. The contributions were aimed at fixing existing open issues on the selected open source projects. In total, the students submitted 214 pull requests, and 93 of them were merged.

Chapter 4

Methodology

In this chapter, we look at each research question posed in Chapter 2 and build a plan to answer it.

4.1 RQ1: Model Training and Synthetic Evaluation

RQ1. How can we apply large language models (LLMs) to classify comment consistency?

Our approach to answering this question involves training the previously introduced large language models on the existing dataset of Java examples shared by Panthaplackel et al.[28] We choose this approach for two reasons. First, using the existing dataset offers a direct comparison against the existing state of the art. Second, we argue that this approach is valid because this dataset is based on mined open source projects, which are representative of the code that would be encountered in practice. In fact, Fluri et al. found that 90% of changes to code correctly updated comments too, which suggests that the examples in the dataset are most likely reliable based on the manner of collection.[10]

4.1.1 Comment Selection

Because comments do not follow the syntactic structure of computer code, their diversity is virtually limitless. Therefore, attempting to create a model that understands every single written comment would quickly become untenable. In this section we outline different types of comments to narrow down the specific type we consider in this project. Listing 4.1 showcases each type of comment discussed in this section.

Inline and Function level comments In most programming languages, comments may either be written anywhere within code or associated with entire functions or methods. We refer to the first type as *inline comments* and the second as *function comments*. In Java, for example, this second type includes Javadoc formatted comments. We make this distinction because inline comments introduce the additional problem of determining the scope of a comment. Inline comments often provide brief explanations of specific code fragments, but may not contain enough information to be useful for evaluating consistency. Some inline comments may even be commented lines of code and not contain natural language at all.

In this study, we focus on function level comments because they are always found in the same position and are associated with the entire function, even across different projects and programming languages. This restriction implies that our results are only valid for function level comments. Inline comments, due to their even less constrained structure, may be more difficult to classify.

Previously, Liu et al. applied random forests with manually engineered features to classify changed inline comments as either consistent or inconsistent.[20] These features included, for example, whether the method declaration changed as well as the number of changed statements in the commit. However, Chen et al. found that changing the inline comment scope resolution had a significant influence on

the results.[2] Because we are also studying comment consistency across multiple languages, looking at function comments gives us a similar setting for comparison.

Unstructured and Structured comments In this paragraph we make the distinction between structured and unstructured portions of function level comments. The structured portion of the comment refers to usage of pre-determined documentation annotations such as `@param` in Java and `:param` in Python. By unstructured, we discuss primarily the function comment summary which appears first. This summary is normally a free-form description of the usage or purpose of the function in question and does not need to follow a specific structure. This study is primarily concerned with unstructured comments because structured comments can often already be checked effectively by heuristic approaches and integrated development environments. For example, tools such as Javadoc can verify that the type, order and spelling of these structured sections matches between the function and its comment.

```
def foo(arg1):
  '''Unstructured summary comment that explains the usage of the function (the target of
      this project).

  :param arg1: structured information about arg1
  :returns: structured return value explanation
  '''
  bar = arg1 * arg1
  # Inline comment explaining details of the code
  # TODO: comment that describes missing behavior
  return arg1 + arg1 + bar
```

Listing 4.1: Comment types

Self-admitted technical debt (SATD) Self-admitted technical debt (commonly also known as *TODO comments*) indicate tasks that are missing or incomplete, and may not provide actionable information about the code itself.[50] In contrast with other types of comments that describe behavior existing in the code, these comments will specifically describe what *is not* in the code. Therefore these comments are excluded from the study, as their criteria for consistency is that they describe functionality that does not exist, which is opposite from all other comments.

4.1.2 Dataset

The initial dataset contains a balanced sample of 40,688 examples of labeled code and comment pairs extracted from 1,518 Java code projects on GitHub.[28] For our work, because of our focus on unstructured comments, we consider only the Summary class of the dataset, amounting to 10,498 examples. We follow the original 80-10-10 split of the dataset to obtain train, validation, and test sets. These sets are partitioned so that there is no overlap between projects; all examples from any given project will only be in one of these sets.

Using the CodeBERT tokenizer, which is based on the byte-pair encoding tokenizer used in RoBERTa [19], the median length of examples from the training set is 131, the validation set is 128, and the test set is 123. Approximately 13% of all the examples have a length that exceeds the BERT maximum token length of 512 (including the special `[CLS]` token). This tokenizer has a vocabulary size of 30,000 tokens.[5]

Under the Codegen tokenizer, which has a vocabulary size of 50,000 tokens[26], the median lengths are respectively 98, 97, and 93. We can attribute the difference to the larger vocabulary size of the Codegen tokenizer, which allows each token to consist of more characters, on average.

Classification Rules To classify examples for the dataset, the authors used projects' commit histories. Given a commit that modifies a particular method, if it changes both the code and its comment, then the old comment is not consistent with the new code. If a commit modifies the code without modifying the comment, then they label the old comment as being consistent with the new code.

```
47  47      /**
48      -    * Maximum block size.
    48  +    * Maximum block size in bytes.
49  49       * @type {number}
50  50       * @constant
51  51       */
52  52      static get BLOCK_SIZE_MAX() {
53      -        return 5e5; // 500 KB
    53  +        return 1e6; // 1 MB
54  54      }
55  55
```

Figure 4.1: Example of a commit that modifies both the code and the comment

In Figure 4.1, we show an example of a commit that modifies both the code and the comment for a function[1]. The modification adds the text `in bytes` to the comment and changes the returned value from `5e5` to `1e6`. Following our classification strategy, the comment before the commit would be labeled as *inconsistent* with the new code.

One result of this labeling strategy was a significant imbalance in the obtained dataset. In practice, it was significantly more common that code was modified without changing the comment than the alternative. Thus, the authors obtained many more examples of the negative label than the positive. To address this issue, Panthaplackel et al. downsampled the negative case to the number of positive examples.

Post-hoc and Ad-hoc evaluation Panthaplackel et al. [28] introduced two classes of comment consistency detection: *post-hoc* and *ad-hoc* (or just-in-time). In the post-hoc setting, the model only receives the final comment and code pair and must determine whether they are consistent. In the ad-hoc setting, the input to the model also includes the previous value of the comment and code, which is assumed consistent, and the model should predict whether the new pair is still consistent. This setting corresponds to the use case of detecting inconsistencies at commit time or while editing. In our study, we mainly consider the post-hoc setting because a model capable at this task would automatically succeed in the ad-hoc setting as well simply by ignoring the code changeset and only looking at the post-change code.

4.1.3 Model Selection

The field of large language models is advancing quickly, with new models and techniques progressing the state of the art at an accelerating pace. [51] In this work, we focus on two pre-trained language models that represent a broad spectrum of current research.

- **CodeBERT** applies additional code-specific training on top of the original BERT language model. [9] This bidirectional encoder model has shown strong results on downstream tasks including code classification, which are closely related to the goal of this project.

[1] source: `https://github.com/nimiq/core-js/commit/b4d0fc32b4c6c1b5ca4c2c127f64eec669a116ad`

- **CodeGen** is an open source model built to offer an alternative to OpenAI's closed source Codex model.[26] This model, based on a transformer decoder architecture (like GPT), was trained on several gigabytes of code and demonstrates strong capabilities in code generation tasks.

To apply large language models (LLMs) for comment consistency classification, we fine-tune the CodeBERT and CodeGen models. Chapter 3 discussed details of these models including their training parameters and previous results.

Specifically, for CodeBERT, we obtain a final classification by attaching a dense layer with two outputs to the final output embedding of the unique `[CLS]` token which was prepended to the input. This follows the approach the original authors used for their evaluation on downstream tasks.[9] This model is illustrated in Figure 4.2.

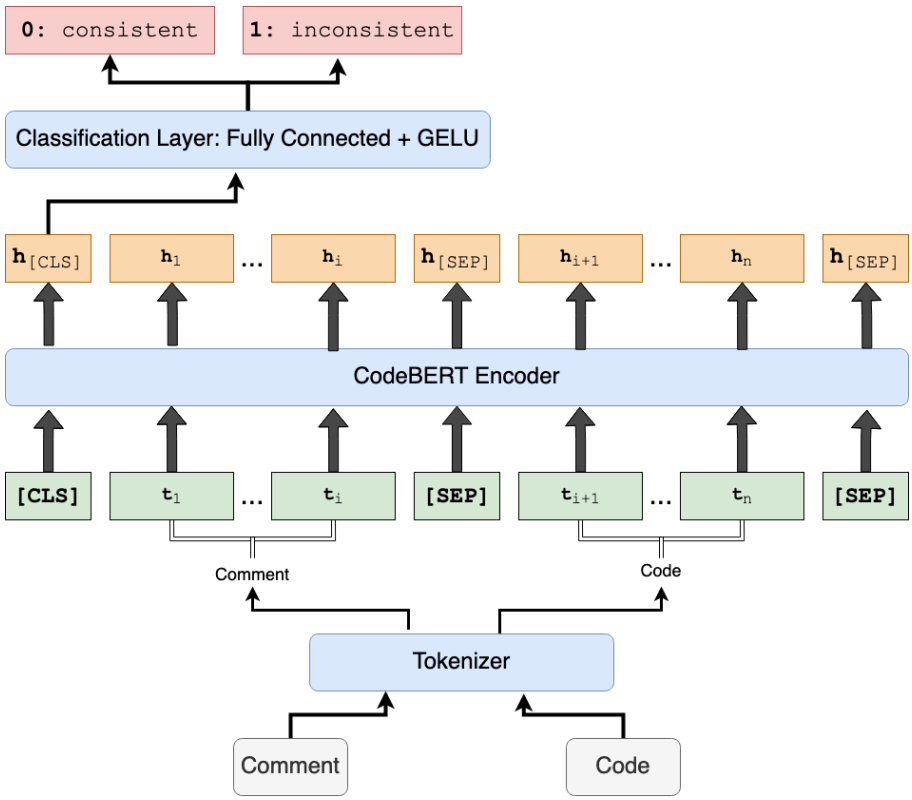


Figure 4.2: CodeBERT model diagram

CodeGen, as a decoder model, contains a final output layer that produces a probability distribution over the entire token space. For our fine tuning approach, we replace this layer with a binary output dense layer, corresponding to our two class label space. We include a diagram of this model in Figure 4.3.

To test how model size scaling affects the results, we incorporate two versions of the CodeGen model: 350 million and 2 billion parameters. We include this test because past works have shown that model size is a significant factor in the performance of LLMs.[17] However, larger models usually require more data to achieve the expected improvement in performance, a condition which this dataset may not satisfy. Therefore, we test whether the larger model size is able to improve performance on this dataset. If the larger model does not improve performance, then users of this dataset can lower computational costs by training smaller models.

Training Details Replicated models, namely DeepJIT, BERT, and Longformer, were trained with identical hyperparameters to their original publications. Where possible, the same code was used via their

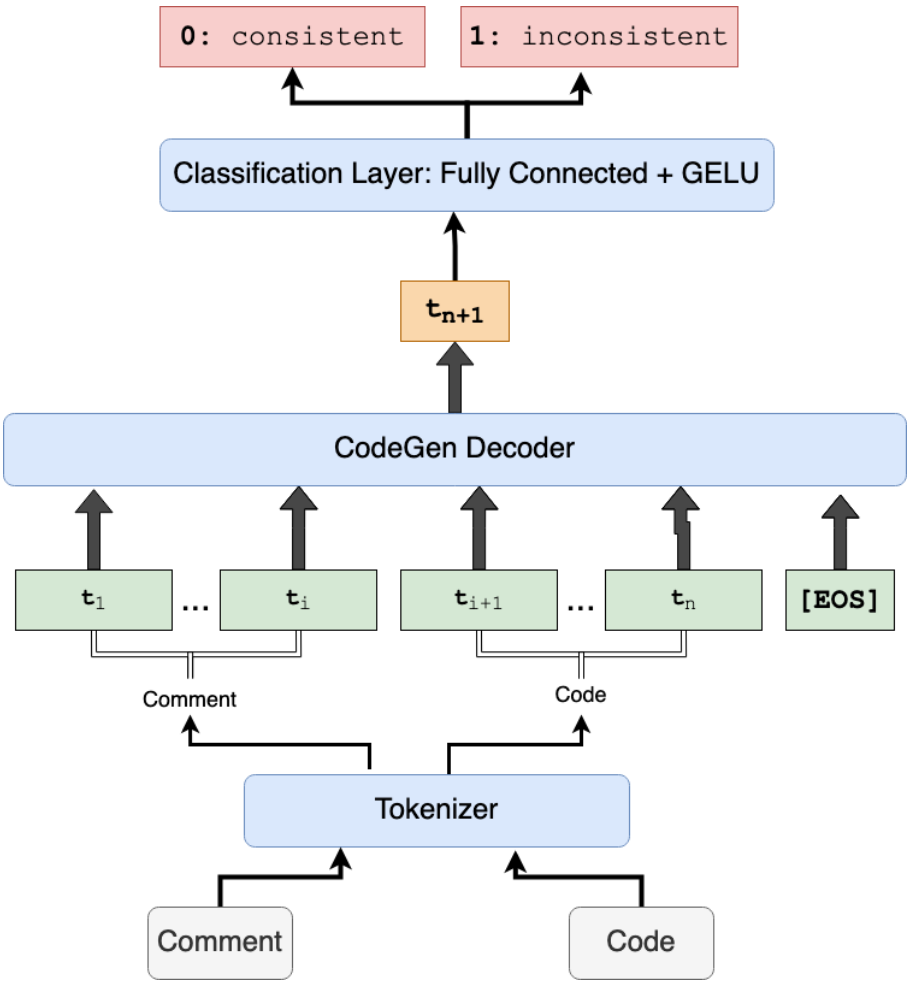


Figure 4.3: CodeGen model diagram

published replication packages. In the case of DeepJIT, only the base model without the Java AST-based graph neural network module was used. We chose this approach because this part of the network could not be readily adapted to work with other languages.

The new CodeBERT and CodeGen models were each fine tuned on a single Nvidia V100 processor. We used a learning rate of 10^{-5} for CodeBERT and 10^{-6} for CodeGen. Each model was trained for 20 epochs and the model with the highest F1 score on the validation set, evaluated at the end of each epoch, was saved and tested.

4.1.4 Evaluation

We compare our models against the replicated past works of Panthaplackel et al.[28] (DeepJIT) and Steiner et al.[39] (BERT and Longformer). In total we therefore perform 6 experiments: CodeBERT, CodeGen 350M, CodeGen 2B, DeepJit (without AST), BERT, and Longformer.

Synthetic evaluations of past works have focused on the F1 score, defined in Equation 4.1.

$$F_1 = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.1}$$

While this score does give a balanced insight into the performance of the model, we suggest that it is not the most accurate reflection of the comment consistency detection setting.

Due to the class imbalance inherent to this problem, we instead evaluate results using a *weighted F1 score*, with class weights derived from the frequency of class labels found during dataset creation. In practice this results in approximately a 19:1 ratio of class weight between the negative (consistent) and positive (inconsistent) labels. In other words, this metric will more heavily penalize the model for making a false positive misclassification, where it decides a consistent example is positive. One advantage of this approach is that a basic classifier which returns "1" for every input would no longer achieve near state of the art performance. In the balanced setting, such a classifier has perfect 1.0 recall and 0.5 precision, for an F1 score of 0.67, which offers strong competition against the current state of the art score of 0.72. Using the weighted score, this baseline classifier's score decreases to 0.1.

We demonstrate this effect in Table 4.1 and Table 4.2. Table 4.1 shows the calculation of F1 score when the class sizes are balanced, which matches the results reported in prior work. Table 4.2 demonstrates the issue that the same predictor achieves a much lower F1 result when the true class ratios are applied to the calculation.

	Predicted Positive	**Predicted Negative**
Actual Positive	50	0
Actual Negative	50	0
Recall: 1.0	**Precision: 0.5**	**F1: 0.67**

Table 4.1: Balanced Confusion Matrix of 100 samples

	Predicted Positive	**Predicted Negative**
Actual Positive	5	0
Actual Negative	95	0
Recall: 1.0	**Precision: 0.05**	**F1: 0.1**

Table 4.2: Confusion Matrix with 100 samples, 20:1 negative:positive ratio

Another advantage of this metric is that it brings our evaluation into alignment with key findings from industry-standard development tooling teams. Sadowski et al., in a study conducted at Google, found that one of the criteria for a useful development tool is that it must produce less than 10% false positives.[31] This false positive rate corresponds to a precision of 0.9. Notably, the recall score seems to be less important for developers' perception of the tool. This insight matches with the weighted F1 score which also strongly incentivizes models which can reach a high precision score.

Qualitative Analysis To gain insights into the LLMs' decision-making process and better understand the models' behavior, we perform qualitative analysis using saliency maps on input tokens. Using the open source visualization tool Ecco, we can analyze attribution scores for each input token based on the output.[1] By leveraging the technique of backpropagation, we can highlight the tokens in the input (comment and code) that contribute the most to the LLM's classification decision.

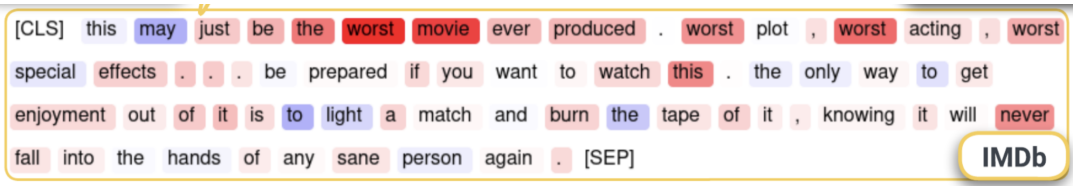


Figure 4.4: Example of saliency map visualization from sentiment analysis [8]

In Figure 4.4, we show an example of a saliency map visualization from a sentiment analysis model. This visualization highlights the phrase "worst movie ever produced", which is the most important

sequence of tokens in the input for the model's classification decision as a negative sentiment.

While we are not able to analyze every weight at every layer of these models, these visualizations can point towards the key patterns in the input that the model has learned. For example, in our data, we may find the most weight placed on variable names in Java when the mistake in the comment relates to a mismatch between a variable named in the comment and its usage in code.

The goal of these visualizations is to understand *why* the models made the predictions they did. Because the Ecco visualization tool is suitable for decoder transformer models, as input we used the Codegen 350M model trained on the original dataset. As a minor technical note, for compatibility with the tool, we pretend that the model is a full-fledged decoder with a full output space of 51,200 tokens, though in reality we only have nonzero probabilities for the first two token classes corresponding to negative and positive.

4.2 RQ2: Multilingual Data Collection

RQ2. How can we evaluate consistency on languages outside of Java?

Our methodology for this research question involves collecting new datasets in Java, Python, Go, and JavaScript and then evaluating our models for each.

4.2.1 Dataset Collection

Project Selection Naturally, the commenting practices of someone writing code for a small personal project may differ greatly from those we would expect in an industry setting. Therefore, we must be very selective about which repositories to include in our study. For our work, we considered the following factors when creating our datasets. Our choices of thresholds are based on the advised peril avoidance strategies of Kalliamvakou et al.[16] Where concrete threshold advice was not given, we derived suitable values from analysis of a recent dataset of GitHub repository metadata.[6]

- Number of stars: the project should have at least 50 stars. As previously mentioned, star count alone is not necessarily adequate for identifying engineered projects. However, it is still the single strongest indicator, where stargazer count alone gives a 97% precision in identifying such projects.[25]

- Number of authors: the project must have at least 10 different people who have committed code, thus avoiding most personal projects. Kalliamvakou et al. noted that 95% of repositories on GitHub had 3 or fewer committers. By looking for at least 10 committers to a project, we limit the scope of our dataset to codebases that are more suited for contributions. Our implicit assumption is that such projects have a stronger motivation to keep comment quality high, and therefore they constitute ideal sources of data for our purpose. Out of all projects with at least 50 stars, 8.8% also had 10 committers.

- Recent activity of project: the latest commit should be within the last 4 years. For applicability to current software development practices, we use this criterion to filter out possibly outdated code patterns. Most projects that passed the previous filters would satisfy this criterion as well. 80.5% of all repositories with at least 50 stars had also been committed to since 2019.

- License type: we will only include permissively licensed projects (such as MIT license). This restriction enables further research and sharing of code without undue restrictions.

- Pull request usage: we will only look at projects with at least 50 pull requests, an indicator that a project seeks external involvement and thus an incentive to keep comment quality high. Among projects with at least 50 stars, 19.1% also fulfilled this rule.

Classification and Risk Mitigation We follow the definition of Panthaplackel et al. in how we label comments as either consistent or inconsistent.[28] However, one risk of this type of automatic labeling is that we might include false negatives in the dataset. In other words, the goal of the model is to detect comments that are inconsistent, but there is nothing to explicitly prevent already-inconsistent comments from being labeled as consistent in our dataset and thus contaminating the results. We take two steps to mitigate this risk potential:

1. Our selection of projects as described in the previous section to include should filter out most low quality code.

2. After creating the datasets, we randomly sample 100 negative examples for inspection. Although we are not experts in every represented codebase, we can interpret obvious mislabelings, such as a comment that was clearly copied from an incorrect place.

Language Specific Details To parse the required comments and code from each language we study, we leveraged the parser generator tool `tree-sitter` to extract functions and their documentation comments. In our problem setting, we parse out the documentation comment and define the *summary* as the contents of the comment up to either the first period, the first empty line, or the first structured directive (e.g. `@param` in Java). Following previous work, we also only include functions where at least one return statement or the return type was changed between the old and new versions, since the documentation summary often relates to the output of a function. Listing 4.2 shows an example of a function with its documentation comment before the change and the Listing 4.3 shows the same function after the change. This example would be included in the dataset because it satifies the requirements that we have defined.

```
/**
 * Returns the sum of the two inputs.
 * @param a the first input
 * @param b the second input
 * @return the sum of the two inputs
 */
public int operate(int a, int b) {
    return a + b;
}
```

Listing 4.2: Before change example

```
/**
 * Returns the product of the two
     inputs.
 * @param a the first input
 * @param b the second input
 * @return the product of the two
     inputs
 */
public int operate(int a, int b) {
    return a * b;
}
```

Listing 4.3: After change example

Listing 4.4 and Listing 4.5 show another pair of functions. However, this pair would not be included in the dataset because only the the parameter documentation lines beginning with `@param` have changed in the documentation comment. For our study of unstructured comments, we only consider the summary of the comment, so this pair would not not satisfy the requirements.

```
/**
 * Returns the sum of the two inputs.
 * @param a the first input
 * @param b the second input
 * @return the sum of the two inputs
 */
public int operate(int a, int b) {
    return a + b;
}
```

Listing 4.4: Before change example

```
/**
 * Returns the sum of the two inputs.
 * @param input1 the first input
 * @param input2 the second input
 * @return the sum of the two inputs
 */
public int operate(int input1, int
    input2) {
    return input1 + input2;
}
```

Listing 4.5: After change example

Filtering To improve the quality of our datasets and reduce noise that would affect our results, we follow several steps in a pre-processing and filtering pipeline.

- Deduplication: we remove all exact copies of the same comment and code after the first one. Examples of this issue could occur when blocks of code were copied and pasted between locations. We also observed this rarely across different repositories when source code from a dependency project was included within the dependent project and periodically updated.[2]
- Generated code: we define a source code file as possibly generated if the word `generated` appears in the first 100 characters or in its file path. Our dataset removes all examples from these files. Popular code generation tools such as `controller-gen` and `grpc` will by convention leave the text `generated` either in the file path or in the first lines of the generated file.
- Deprecations: we remove examples where the word `deprecated` appears, as this decision is made outside of the code base and cannot normally be determined from the contents of the function.
- Whitespace only changes: we remove any examples where the only difference between old and new is whitespace. This happens frequently when projects change formatting tools or migrate from tabs to spaces.
- SATD: as discussed in the problem formulation, we remove examples where the comment is likely self-admitted technical debt (e.g. containing `TODO` or `FIXME`). We do this because the consistency of SATD is expected to be opposite of other comments. The technical debt comment is consistent if it is accurately describing something that does not exist in the code. Otherwise, if the code implements the described debt, then the comment is inconsistent

We include several examples of these filtered cases in Appendix A.1. We created training, validation, and testing sets following a split of 80, 10, and 10 percent.

4.2.2 Modelling Decision

In this study, we must decide between two approaches for model training on multiple languages. Either we train each model on each language separately (for each type), or we train a single model on examples from all languages. In our test, we compared a model trained on all languages to a model trained on only Python, evaluated on the Python data test set and following the training details given in section 4.1.3. The results of this test are shown in Table 4.3.

	Precision	**Recall**	**F1**
CodeBERT (trained on Python)	0.62	0.49	0.55
CodeBERT (trained on 4 languages)	0.74	0.59	0.66

Table 4.3: CodeBERT trained on all 4 languages or only Python, evaluated on Python test set

While it might seem intuitive that a model trained on multiple languages will sacrifice performance compared to a specialized model, we find in our preliminary experiments on Python that a model trained on all examples achieves a better result. One possible explanation for this behavior is that using a single model benefits from having a larger and more diverse dataset. By combining data from multiple languages, we increase the overall volume of training examples, potentially leading to better generalization and a more robust model. Therefore, all results in this work for **RQ2** are produced by models that have been trained on all 4 languages.

4.2.3 Evaluation

As discussed in subsection 4.1.4, we compare models based on weighted F1 scores, categorized per language. After collecting the multilingual datasets in the manner discussed in this section, our first experiment determines whether training new models is necessary by testing whether a model trained

[2]This is known as *vendoring*

only on Java can generalize to other languages by showing equal performance on unseen languages. If not, then we train all of the models on the new languages as well, and we will compare their results on the multilingual test set after training. We will also perform qualitative analysis by investigating specific examples and visualizing learned patterns through input saliency.

4.3 RQ3: Practical Benchmark Creation

RQ3. How do we create a benchmark that matches real world situations?

In this section, we describe our approach to manually curating examples for a benchmark set that mimic real-world usage as closely as possible.

4.3.1 Pull Request Comment Mining

In correspondence with Wen et al.[44], who authored a large-scale empirical study of commit messages, we proposed a similar technique of looking for specific patterns within pull request comments to construct a benchmark set. Our hypothesis was that we could find effective examples of real-world situations by looking at instances where code reviewers specifically asked code authors to modify comments to improve consistency with code.

They agreed with our intuition that if a maintainer writes a review comment on a pull request asking the author to *fix* or *update* a comment, that gives us a strong signal that the referenced text is a real world example of an inconsistent comment. These findings point toward the feasibility of this method for constructing the benchmark set to answer **RQ3**. We discuss the results of this pull request comment analysis in Chapter 5. We also highlight the challenges of this approach in Chapter 6.

Our method for finding these real-world representative situations was to mine a large number of comments from the GitHub GraphQL API, apply some heuristic filtering, and then manually inspect each example to see if it could enter the benchmark set. Our procedure was as follows.

- Mine several million pull request comments (details shared in Chapter 5).
- Filter for comments attached to lines of code with text containing a comment-relevant word (e.g. `comment`, `document`, `javadoc`) and a relevant action verb (e.g. `fix`, `update`, `outdated`).
- Manually filter the resulting examples and include those that 1) were fixed, 2) were applied to a function or method, 3) modifies the unstructured portion of the comment to match our problem scope.
- To reduce bias in the benchmark set, we only allow one case from any given repository.

The goal of this approach was to find approximately 100 examples, 25 for each studied language. Because each example contains a before and after with known labels, where the before is inconsistent and the after is consistent, this process would give us 200 samples on which to benchmark our models. Due to the large amount of manual work involved in preparing these examples, finding yet more samples was out of scope of this project. However, the value of 200 samples is in line with previous benchmark sizes in the field. HumanEval, a popular code generation benchmark, contains 164 programming problems.[3] MTPB, as discussed in the background research in Chapter 3, contains 115 separate examples.[26]

We found that for JavaScript (even after including TypeScript), looking for pull request comments alone did not produce enough examples. In this case, we augmented the process by including commit messages in the search. We followed the method of Wen et al.[44] by applying the same heuristic of finding messages containing a comment-relevant word and an action verb. With this approach we were able to reach the target number of examples for all languages.

Each example of an updated comment indicates an old version and a new version, where the old version was corrected to the new version as a result of the review comment on a pull request. Thus, we label the old version as positive (inconsistent) and the new version as negative (consistent) because it passes an expert code review. In total, we found a balanced set of 100 examples of inconsistent

comments, leading to 200 benchmark cases by splitting positive and negative labels, meaning a final list of 50 per language.

Subjectivity Risk Mitigation We define subjectivity risk as the potential for bias in the benchmark set as a result of one person performing all of the labeling. Although we have written specific criteria for the method, there remains room for interpretation in deciding whether a specific comment indeed asks for revision. To mitigate this revision, we defined a sample set of 16 Python cases from the result of automatic heuristic filtering. We recruited 3 individuals along with the main author of this project to label these examples, and we compared the resulting benchmark cases between them for correspondence. If there was substantial disagreement, we would revise the procedure for finding and determining the cases to reduce ambiguity.

4.4 RQ4: Real-world Performance

RQ4. How do developers react to the results of this model?

The goal of answering this research question is to discover how programmers respond to the usage of this model in a social environment through GitHub pull requests. Under the taxonomy of Stol et al.[41], this work would be classified as a *solution-seeking* study, with the aim of solving the practical problem of inconsistent comments and measuring the impact of this solution. In this work, the actors are software developers in the studied languages, the measured behavior is response to pull requests, and the context is open source development on GitHub.

We choose to use GitHub as the setting for this study because of the popularity of the platform, where we can reach maintainers of different organizations for all of the studied languages. To perform this part of the study, we will execute our classification on selected open source projects. We select open source projects that have a minimum number of 25 stars and at least three pull requests submitted in the studied time period. The star count threshold is used to focus on engineered projects.[25] The recent pull request submission rate is used to filter for active projects. This step increases the likelihood that a maintainer will respond to our submission.

After the model identifies a potentially inconsistent comment, we manually review the issue. If we agree, then we will do our best effort to write a fix by hand and submit it as a pull request. Our goal is to have approximately 20 responses to our pull requests. Because of the high touch nature of this work we cannot indefinitely submit requests. In other words, each pull request requires a significant amount of manual work to write a relevant fix and submit it to a project on which we have no prior context.

However, we believe that 20 responses suffices to draw conclusions from this project because this process corresponds to the ideal use case of this model. This setting emulates the situation where a programmer introduces an inconsistent comment and is immediately informed of the fact by a theoretical perfectly accurate model. This usage is the ideal situation for our model, and the results here can answer the question of whether comment contributions are helpful or wanted because they are of the highest possible quality. Finally, we support this choice by prior related work. In a related study on comment consistency, Tan et al. sent pull requests to fix 16 identified inconsistencies, where 5 were responded to by the developers.[42] Figure 4.5 shows an example of the pull request template we use.

In addition to the proposed fix, we also attach a survey in the original request form which has received approval from the ethical review board[3] at TU Eindhoven. This survey asks for reasoning behind the accept or reject decision and includes questions to rank the relative significance of comment consistency against other issues such as complex class hierarchies and meaningless variable names in terms of developer frustration in *code comprehension* activities. The exact list of potential factors is taken from the findings of a large scale field study by Xia et al., where the authors found that insufficient comments were a significant source of frustration.[47] However, we expect a potentially low response rate to this survey. In a recent work that directly emailed survey questions to software maintainers, Xia et al. (distinct authors from the previous paper) sent 191 emails but only received a response rate of

[3] approval id: ERB2023MCS11

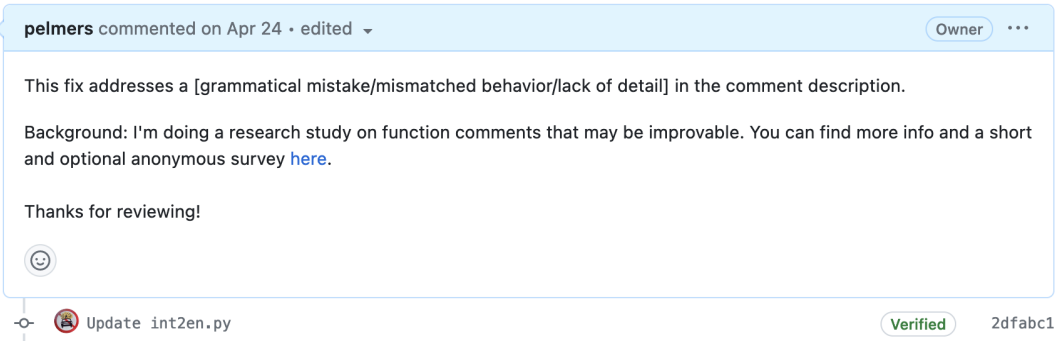


Figure 4.5: Example pull request associated with **RQ4**

17.8%.[46] Our survey faces additional challenges. Not only is it not directly addressed at maintainers, but the survey questions require a consent to a verbose ethical and legal statement.

4.5 Ethical Considerations

Existing language and code generative models face strong headwinds of ethical and legal concerns. For example, they may produce code that looks like licensed work because of oversights in the training data collection process. Professional developers may be concerned about the future of their job security because they see generative models as direct competition. And these models may be employed by malicious actors to build programs that exploit security vulnerabilities or target other people.

However, in this work, our discriminative models avoid most of these concerns. Our model's only output is a binary determination of consistency. There remains an ethical concern that developers could rely too heavily on this model and allow their own attentiveness to lapse. The psychological theory of *risk compensation* states that people adjust their behavior to maintain an expected level of risk. If developers feel safely protected by this model, then they will proportionally decrease their own effort and thus the final quality of comments in code could stay unchanged. The purpose of the social study in **RQ4** is to slightly mitigate this risk by studying how programmers would react to usage of this model.

Another possible ethical impact of our work is the increased burden on software maintainers as a result of reviewing our contributions for **RQ4**. To mitigate this effect, we only submit at most one pull request per project. Overall, the ethical risk of this study is minimal because we do not study personal information on a large scale, surveil people systematically, or make decisions that will affect people.

Chapter 5

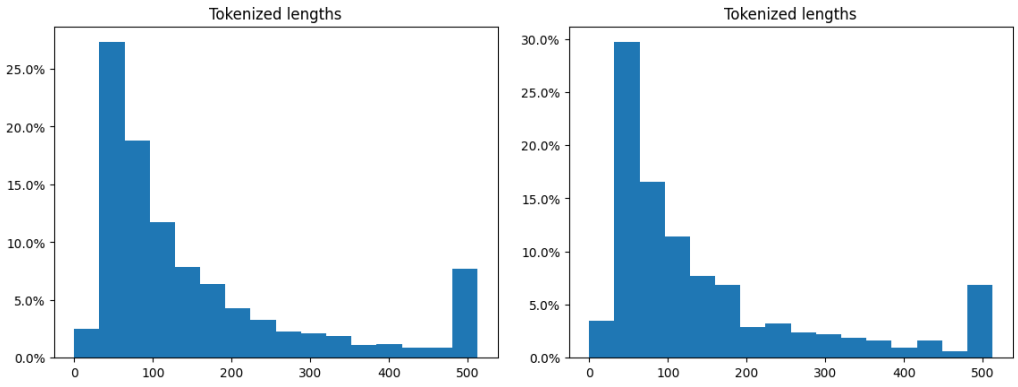
Results

This chapter presents the main results of this project, grouped by each research question. It also contains an analysis of the most important findings.

5.1 RQ1: Tests on Existing Java Dataset

As described in our section on methodology in Chapter 4, our approach to answering this question involves training machine learning models on the previously published Java dataset of Panthaplackel et al.[28]

Input Truncation Note that Longformer has an input maximum length of 2048 tokens. All other models in the test used an input length of at most 512 tokens, with longer examples truncated at that point. We estimated the effect of this truncation by calculating statistics over the lengths of the training set. For instance, if the number of truncated examples would be very large, then we would expect model performance to be negatively impacted. However, of the 8,398 training samples, only 585 had a combined token length (sum of code and comment lengths) greater than 512. The mean length in the training set is 304 tokens, with a median of 130. Thus, most samples will not be affected by limitation in the input length of the model. Figure 5.1 plots the length distribution of examples in the training and validation sets.



(a) Training lengths

(b) Validation lengths

Figure 5.1: Histograms of length distributions in training and validation sets of replication data

Metrics The performance metrics of each model on this dataset are shown in Table 5.1. The weighted F1 score is calculated by applying a sample weight of 19 to all negative examples and a sample weight

of 1 for all positive cases. These weights reflect the proportions of these classes' occurrences in the original, pre-downsampled, datasets, rounded to the nearest whole number.

	Precision	Recall	Unweighted F1	Weighted F1
DeepJIT Base[28]	0.796	0.527	0.634	0.272
BERT[39]	0.582	0.629	0.605	0.133
Longformer[39]	0.866	0.835	**0.850**	0.409
CodeBERT	**1.000**	0.704	0.826	**0.827**
Codegen 350M	0.995	**0.734**	0.845	0.816
Codegen 2B	0.997	**0.734**	0.846	0.817

Table 5.1: **RQ1** Test Performance Comparison

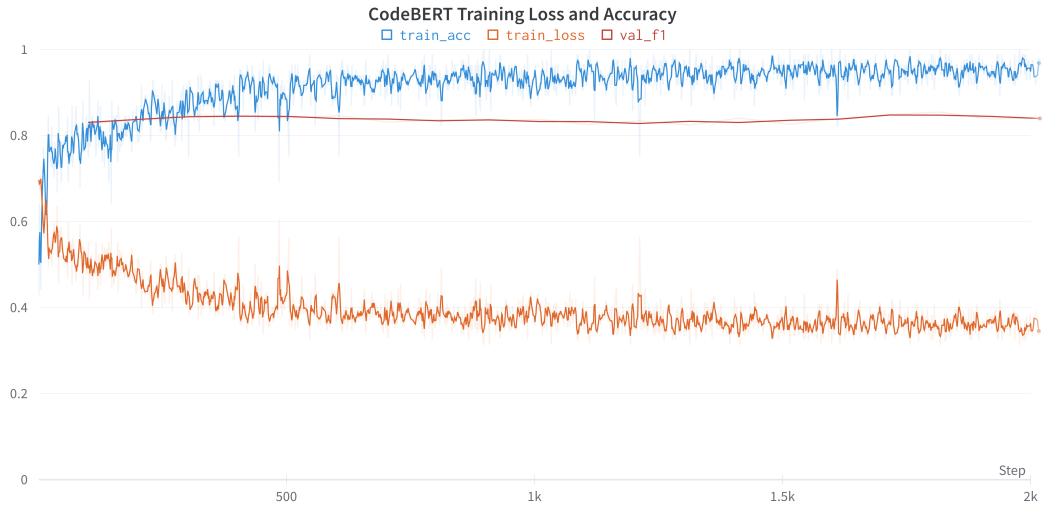


Figure 5.2: CodeBERT Training Loss and Validation F1, each epoch measured in 100 steps

5.1.1 Analysis

From Table 5.1, we see that the new models outperform previous models by weighted F1 score. In particular, we can see that the precision of these models is almost perfect, perhaps too good to be true. That does turn out to be the case. Overall, the best model in this test was fine-tuned CodeBERT. We share the training loss, accuracy, and validation F1 scores during training for CodeBERT in Figure 5.2. This graph shows that accuracy and F1 scores seem to reach a plateau very early on during training.

Our replicated models perform similarly to originally published results. Specifically, in this setting, unweighted F1 scores were previously used. Panthaplackel et al. reported an F1 score of 0.706 for their model. Steiner et al. reported F1 scores of 0.72 and 0.84 for their BERT and Longformer models, respectively.

Visualizations of Model Performance To explain the results we have seen, we used the Ecco library to visualize the input saliency of the models on selected input examples.[1]

Figure 5.3 illustrates the normalized contribution to the final classification by token for an example from the replication dataset. Note that the greatest contribution by far comes from the final token, a pair of newline characters following the method body, which accounted for 15.16% of the total. To test whether the model was tuned to fixate on this feature, we removed the trailing new lines and visualized the result in Figure 5.4. To our surprise, we saw the outcome class change from positive to negative and a new gradient distribution.

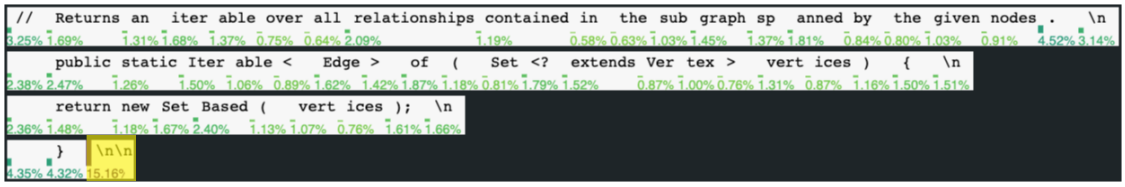


Figure 5.3: Example Gradient Norm Visualization, trailing newlines highlighted

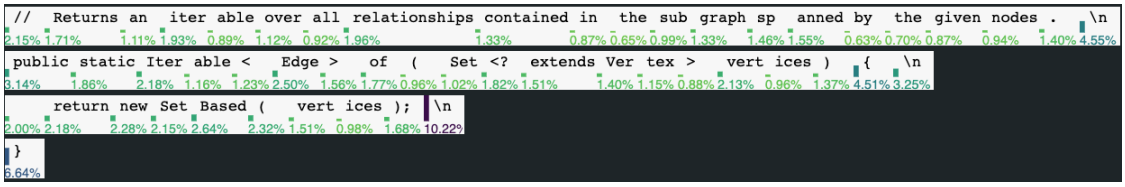


Figure 5.4: Example Gradient Norm Visualization, trailing newlines removed

From this finding, we built a very simple trailing new line based classifier on the replication dataset. This classifier simply returns the positive class if it sees two newline characters at the end of the input. Otherwise, it returns the negative class. We express this classification rule in Equation 5.1.

$$\text{prediction} = \begin{cases} \textit{inconsistent}, & \text{if example has 2 trailing new line characters} \\ \textit{consistent}, & \text{otherwise} \end{cases} \tag{5.1}$$

This simple model in fact achieves an *identical* result to our 100% precision trained CodeBERT model on the replication data. Therefore this model also achieves leading accuracy and F1 on the replication dataset because *all positive examples* in the dataset contained two trailing new line characters, including the training, validation, and test sets. Additionally, approximately 70% of negative (consistent) examples did not have two trailing newlines, which allows models to learn this difference during training and use it during classification.

5.2 RQ2: Extension to Other Languages

The goal of **RQ2** is to investigate how well the methods that seem to achieve strong results on the original Java dataset will perform on other languages. To answer this question, we first collected new datasets for the four studied languages Java, Python, Go, and JavaScript. Following the steps in our methodology, we tested how well a Java-trained models would evaluate on the new languages, then trained new models on the new languages for comparison.

5.2.1 Dataset Details

For each language, we mined examples from GitHub public repositories. The statistics on the number of examples and repositories are given in Table 5.2. After this mining process, we balanced the overall dataset by keeping 22000 examples for each languages, resampled to achieve an exactly even balance between the positive and negative class. The projects mined were filtered from a total set of 3,152,515 projects using the criteria listed in Chapter 4.

Listing 5.1 gives an example of a positive case for the Python language. In this example, for the given method, both the code body and the comment text were changed in a single commit, shown in Listing 5.2. Therefore, we label the old comment as inconsistent with the new code. Here, from inspection, we see that the inconsistency lies in the change from `id` to `key` for the database query.[1]

[1] `https://github.com/medtagger/MedTagger/commit/9a6d61066a7dda328dd9552e1c05cab57cd42f0c`

Language	Number of Projects	Examples	Positive Examples
Java	2257	259,215	15,066
Python	3940	279,440	16,737
Go	2982	573,044	29,779
JavaScript	13,981	296,092	11,059
Total	23,161	1,407,791	72,641

Table 5.2: Dataset Statistics

```python
# Comment: Fetch all tasks from database
    ordered by id.
def get_all_tasks() -> List[Task]:
    with db_session() as session:
        tasks = session.query(Task).
            order_by(Task.id).all()
    return tasks
```

Listing 5.1: Python Example (Before)

```python
# Comment: Fetch all tasks from database
    ordered by key.
def get_all_tasks(include_disabled: bool
    = False) -> List[Task]:
    query = Task.query
    if not include_disabled:
        query = query.filter(~Task.
            disabled)
    return query.order_by(Task.key).all()
```

Listing 5.2: Python Example (After)

Table 5.3 and Figure 5.5 display statistics on the lengths of the examples in the training set by language in terms of number of tokens after tokenization using the CodeBERT tokenizer.

	Mean	Median
Java	170	110
Python	292	201
Go	328	210
JavaScript	293	185

Table 5.3: Mean and Median Dataset Lengths (number of tokens under CodeBERT tokenization)

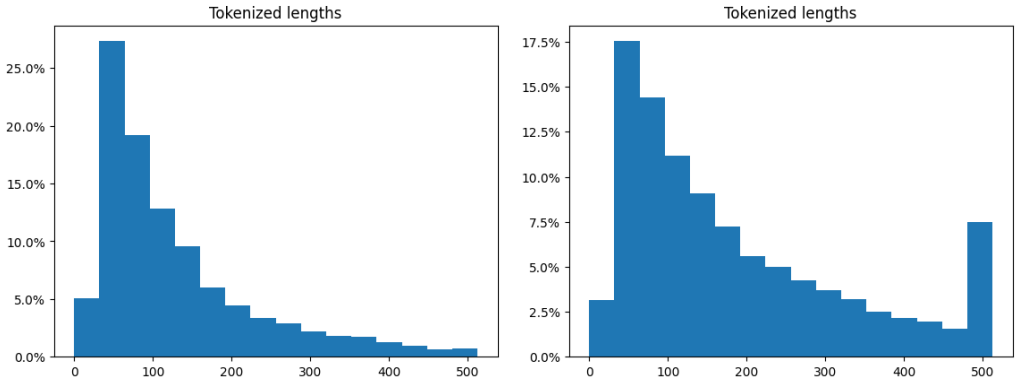
Java One-Liners Note that the lengths of examples in Java were apparently significantly lower than the other languages. In fact, *7721* of the 17600 examples (44 percent) in the training set only had one statement in the method body. One explanation for this result is that Java programs rely heavily on redirection and method overloading, resulting in many methods which simply return the result of calling another method. We give an example of a one-line method in Listing 5.3. This observation is interesting because shorter methods give less context for the model to make a classification decision. Thus, we may expect that the model will have more difficulty to classify Java examples than other languages.

```java
// Comment: Calls #createConfigStore(
    ServiceSpecification, Collection)
    with an empty list of custom
    deserialization types.
public static ConfigStore<
    ServiceSpecification>
    createConfigStore(
        ServiceSpecification
            serviceSpecification) throws
            ConfigStoreException {
    return createConfigStore(
        serviceSpecification, Collections
        .emptyList());
}
```

Listing 5.3: Java One-Line Example (Before)

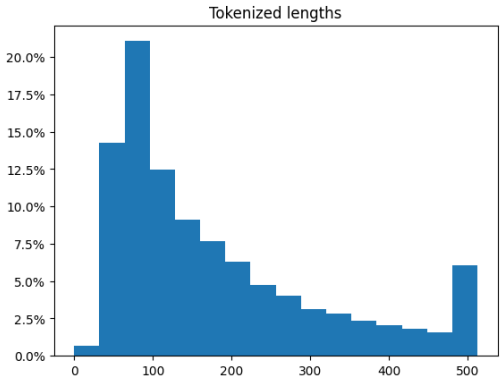
```java
// Comment: Calls #createConfigStore(
    ServiceSpec, Collection) with an
    empty list of custom deserialization
    types.
public static ConfigStore<ServiceSpec>
    createConfigStore(
        ServiceSpec serviceSpec) throws
            ConfigStoreException {
    return createConfigStore(serviceSpec,
        Collections.emptyList());
}
```

Listing 5.4: Java One-Line Example (After)

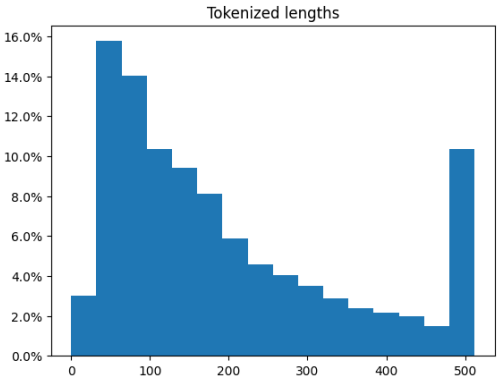


(a) Java lengths

(b) Python lengths



(c) Go lengths



(d) JavaScript lengths

Figure 5.5: Histograms of length distributions in training sets by language

Whitespace Investigation In the results of **RQ1**, we discovered that all inconsistent examples in the original dataset had two trailing newlines. To test whether this was inherent to the data and therefore an accurate predictor of the inconsistency of a comment, we investigated the distribution of trailing newlines in the newly collected training set. In the newly mined GitHub dataset, we tested this by also counting the number of trailing newlines after each example according to its label. Our results are shown in Table 5.4. Here we see no significant correlation between the label and the number of newlines following the method. In the negative case, approximately 10% of examples have one newline. In the positive case, that ratio is approximately 8%.

Number of trailing newline characters	**1**	**2**	**3+**
Negative	19,653	213,401	10,585
Positive	991	13,581	456

Table 5.4: New Line Results in Java

Therefore, we can conclude that the wide gap in performance between the newly mined Java dataset and the original Java dataset comes down to this artifact in the original data collection process. It is unclear what caused this annotation artifact in the dataset. Our best guess is that it arose from a bug in the data collection process, where the method body parser could have been changed between collecting the negative and positive examples, so that it included extra newlines in the positive examples, but not in the negative examples.

5.2.2 Necessity of newly trained models

A reasonable test to validate before training models on the large amounts of new data is to evaluate the previously best-performing model on the test sets from the new datasets. If the performance is similar across the new languages added, then we could conclude that our Java-trained model already generalizes well across other unseen cases. For this test, we chose the best-performing CodeBERT model from Table 5.1 and evaluated it on the new datasets presented in Table 5.2. The results of this test are displayed in Table 5.5.

	Precision	Recall	F1	Weighted F1
Java	0.80	0.16	0.28	0.17
Python	0.71	0.04	0.07	0.06
Go	0.79	0.01	0.03	0.03
JavaScript	0.86	0.02	0.03	0.03

Table 5.5: CodeBERT trained on replication data, tested on new data

	Precision	Recall	F1	Weighted F1
Java	0.68	0.54	0.60	0.17
Python	0.59	0.44	0.51	0.12
Go	0.64	0.35	0.45	0.14
JavaScript	0.60	0.33	0.42	0.12

Table 5.6: CodeBERT trained on whitespace-normalized replication data, tested on new data

Due to the trailing whitespace annotation artifact in the original data, we also performed the test by training on corrected data. We normalized the data by removing all trailing new lines from every example, so it was no longer possible to determine a prediction only based on this aspect. We display the results in Table 5.6.

Analysis The results shown in Table 5.5 clearly indicate that the model trained only on Java examples does not generalize fully to other languages, with all other recall scores under 5 percent. Notably, however, even the Java score fails to reproduce the findings from the replication dataset, with a far lower recall score as well. In Table 5.6, we see a much more balanced result for the other languages. However, the results for testing on Java language examples were still significantly better. Even with the corrected artifact, the gap supports the idea that a model trained only on one language would not carry its performance to other languages without any additional training.

5.2.3 Model Results

Now we train each model on the entire dataset. That means we feed all training examples from all mined languages (Java, JavaScript, Python, Go) to each model per epoch, in shuffled order, using the same hyperparameters as introduced in the training setup for **RQ1** in section 4.1.3. To reflect a reasonable balance in the trade-off between precision and recall, we present the weighted F1 score of each model, as defined in subsection 4.1.4. These results are shown in Table 5.7[2].

5.2.4 Model Analysis

Overall Results The results do not point toward a single model as the consistent best performer on the dataset in terms of weighted F1 score. The CodeBERT model, overall, does have the highest score. The base model is pre-trained on a classification task that resembles this setting, as we discussed in section 3.2.3 [9], which likely contributes to its best in class performance in the evaluation stage. Note

[2]DeepJIT model experienced mode collapse: it predicted 1 for all inputs.

	Java	Python	Go	JavaScript	Overall
DeepJIT[28]	0.095	0.095	0.095	0.095	0.095
BERT[39]	0.139	0.137	0.137	0.131	0.138
Longformer[39]	0.152	0.134	0.154	0.149	0.147
CodeBERT	**0.242**	**0.152**	0.203	0.177	**0.197**
Codegen 350M	0.189	0.149	**0.217**	**0.197**	0.189
Codegen 2B	0.195	0.157	0.201	0.187	0.179

Table 5.7: Weighted F1 Scores by Model and Programming Language

that during the evaluation of this research question, the test set on which our results are based was automatically collected in the same manner as the training set. While our methodology dictated that the set of repositories from which examples were mined among training, validation, and test were disjoint, the format and collection method of the examples was identical.

Training Progress We display a chart of loss and accuracy from training of the CodeBERT model in Figure 5.6. In this chart, we can observe the training set accuracy continues to increase and loss decreases throughout training. However, Figure 5.6 also shows a chart of F1 score on the validation set, evaluated at the end of each training epoch. By comparing these charts, we see that the validation set performance stops increasing relatively soon. This gap in performance between the training and validation sets suggests an issue of overfitting.

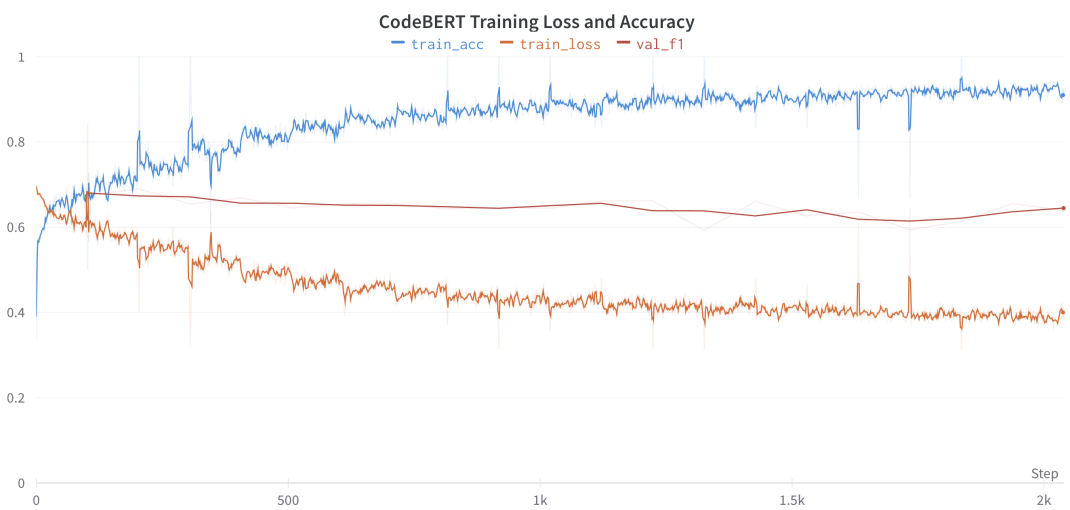


Figure 5.6: CodeBERT Training Loss and Validation F1

Model Size Another piece of evidence for this overfitting hypothesis comes from the relative performance of the new models. We note that the larger models did not necessarily outperform smaller models, despite having many more parameters available. This result suggests that the models have overfit on the training set and can no longer improve overall performance after a certain point, so the extra complexity inherent to the larger models does not produce a benefit in this specific scenario.

5.3 RQ3: Benchmark Evaluation

5.3.1 Overview

The goal of this research question is to evaluate the performance of the trained models on examples extracted from real-world occurrences of code and comment inconsistency. We first created a dataset of manually curated examples, discovered from pull request comments mined with the GitHub API.

5.3.2 Pull Request Comment Details

We list the total number of mined projects, the number of mined comments, and the number of comments after filtering per language in Table 5.8. To filter the list of comments, we applied the following rules.

- Presence of a word related to documentation, such as `JavaDoc`, `docstring` or `godoc`
- Presence of a verb related to an update request, such as `fix`, `update`, or `outdate`
- Existence of a comment block delimiter in the code lines preceding the line referenced by the comment: triple quotes in Python, `//` in Go, and `/**` in Java (for JavaScript we looked for either `//` or `/**`)

	Projects	Total Comments	Filtered Comments
Java	7,094	2,712,741	6,070
Python	15,212	4,418,482	7,600
Go	7,154	3,141,655	8,918
JavaScript	15,608	2,895,875	5,355
Overall	45,068	13,168,753	27,943

Table 5.8: Benchmark Statistics

Despite the aggressive filtering applied to the mined pull request comment data, with only 0.2% of comments passing the filter, we still had a tremendous quantity of examples to manually look through.

In practice, the filtered comments were reduced during the benchmark creation by the additional restriction that we only kept one example for each project. Therefore, when we confirmed one example for the benchmark set, we removed all other examples from the filtered candidates list. Because of this, approximately 800 comment candidates were evaluated for each language.

For each found example, we extracted the review comment that led to the determination and the code blocks before and after a change was made in response. We give an example of one case from the Go language benchmark set in Listing 5.5.

URL: https://github.com/cilium/cilium/pull/2684#discussion_r165204442

Review: Update the documentation for this function to reflect the addition of 'policy.GetPolicyEnabled()' having its result checked.

Old Version:

```
// IngressOrEgressIsEnforced returns true if either ingress or egress is in
// enforcement mode
func (e *Endpoint) IngressOrEgressIsEnforced() bool {
  return policy.GetPolicyEnabled() == AlwaysEnforce ||
    e.Opts.IsEnabled(OptionIngressPolicy) ||
    e.Opts.IsEnabled(OptionEgressPolicy)
}
```

New Version:

```
// IngressOrEgressIsEnforced returns true if either ingress or egress is in
// enforcement mode or if the global policy enforcement is enabled.
func (e *Endpoint) IngressOrEgressIsEnforced() bool {
```

```
  return policy.GetPolicyEnabled() == AlwaysEnforce ||
    e.Opts.IsEnabled(OptionIngressPolicy) ||
    e.Opts.IsEnabled(OptionEgressPolicy)
}
```

Listing 5.5: Example from Go benchmark

Subjectivity Verification We mitigated the risk of subjectivity bias by distributing a random set of 16 filtered Python candidate examples to 4 reviewers. Each reviewer was given the set of examples and a list of instructions on the steps to take to verify the examples. The instructions correspond to a check of each of the criteria listed in 4.3.1. After completion of the labeling process, the results were compared in a discussion.

In this discussion, we reviewed the cases where annotators disagreed on the classification of an example. At this point, from these 16 filtered candidate examples, each reviewer had found between 3 and 6 examples of code and comment inconsistency. During the review, we found that each annotator had made small mistakes in the interpretation of the criteria. For instance, two examples contained comments that were changed, but the change only affected a parameter name, not the summary line. Therefore, the comment should not be included in the benchmark under the criteria set out. However, we found that after discussion, all labelers agreed on the classifications of the examples given in the set. In the reviewed list, the 3 examples that satisfied all requirements were kept as inconsistent and used in the final benchmark set for Python.

5.3.3 Results

After creating the benchmark sets containing 25 real-world examples for each language, we evaluated the models on these examples in the benchmark sets. The results, again using weighted F1 score, are presented in Table 5.9. We omit the replicated DeepJIT model from this comparison because of the mode collapse observed in training.

	Java	Python	Go	JavaScript	Test Set Overall
BERT	0.135	0.097	0.085	0.098	0.138
Longformer	0.156	0.066	0.085	0.078	0.147
CodeBERT	0.103	0.096	0.161	0.122	0.197
Codegen 350M	0.083	0.090	0.078	0.095	0.189
Codegen 2B	0.056	0.110	0.069	0.094	0.179

Table 5.9: Benchmark Set Weighted F1 Scores, compared with test set overall score (from Table 5.7)

Table 5.9 shows a significant decrease in performance from the automatically mined test set to the manually curated benchmark examples.

5.4 RQ4: Social Study Results

Regardless of how well any model performs, the overall goal of a developer tool is to be used. Thus, the objective of this research question was to determine whether the task at hand is useful to real-world developers.

To answer this question, we executed the trained CodeBERT model against popular and active open source projects on GitHub with at least 50 stars and 3 pull requests in the time period of May 15 to May 30, 2023. We manually inspected the model's outputs and identified true positives. We then wrote handcrafted fixes for these true positives and submitted pull requests on GitHub to the project maintainers. In these pull requests, we also included a link to an optional survey which asked for opinions on the usefulness of the tool and the relative frustration of various developmental difficulties.

In total, we submitted 20 pull requests, with outcomes listed in Table 5.10. The full list of pull requests can be found at Appendix A.2.

	Count
Accepted	17
Accepted with Revision	1
Rejected	0
No Response	2

Table 5.10: Pull Request Outcomes

However, we received no responses on the attached survey. In chapter 6, we discuss the implications of our results in the context of open source software research, and in chapter 7 we offer directions for future work that could give more insight from survey questions related to software development through pull requests on GitHub.

5.4.1 Pull Request Analysis

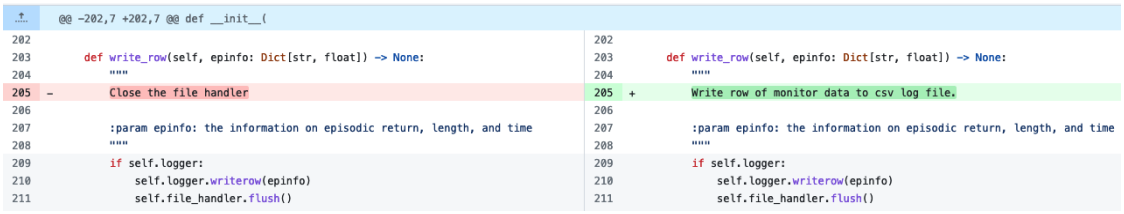


Figure 5.7: Example Comment Fix Pull Request

Figure 5.7 shows an example of an accepted fix to a Python project. Because we are not experts in all of the potential projects on GitHub, we can only confidently write fixes for obvious mistakes. Therefore, even when the model identifies a comment as inconsistent, we usually are not equipped to determine whether that is a true or false positive. Despite this potential limitation, the results in this section suggest that addressing inconsistent comments is a valued contribution to software. From the 90% of accepted fixes and zero outright rejections, we can conclude that these fixes are useful. Unfortunately, without qualitative results on our survey, it is difficult to gauge exactly to what degree the usefulness applies.

Chapter 6

Discussion

This chapter is devoted to interpretation of the results presented in Chapter 5. The goal is to dive into reasons for the results and explore some of the implications of our findings.

6.1 Newly Collected (RQ2) Dataset Details

6.1.1 Dataset Validation

A specific risk of automated dataset collection based on commit history is that we could introduce false negatives to the labeled set. In other words, we may heuristically determine an example as negative, meaning the code and comment are consistent, but actually it should have been labeled inconsistent, or positive. This can happen because this determination is expressly the purpose of building the models, and before we have trained a model the only way to definitively label an example is through asking an expert.

To mitigate this risk, we checked a random sample of 50 examples from each language in the dataset. These examples were manually inspected to validate the correctness of their labeling. The validation process involved reviewing the comments and their corresponding code to determine if the labeling heuristic accurately captured the comment's consistency. Listing 6.1 and Listing 6.2[1] give an example where the source code of a method has changed, but the comment has not. Therefore our heuristic labels the comment as consistent with the new code, and, after manual review, we decided this was an accurate label. This example was one of the 50 from Java we spot checked for correctness.

```
// Return true if an element made up of
    byteables might have been put in this
     filter or false if this is
    definitely not the case.
public boolean mightContain(Byteable...
    byteables) {
    masterLock.readLock().lock();
    try {
        return source.mightContain(Token.
            create(byteables));
    }
    finally {
        masterLock.readLock().unlock();
    }
}
```

Listing 6.1: Java Consistent Comment Example (Before)

```
// Return true if an element made up of
    byteables might have been put in this
     filter or false if this is
    definitely not the case.
public boolean mightContain(Byteable...
    byteables) {
    masterLock.readLock().lock();
    try {
        return source.mightContain(
            Composite.create(byteables));
    }
    finally {
        masterLock.readLock().unlock();
    }
}
```

Listing 6.2: Java Consistent Comment Example (After)

[1] From https://github.com/cinchapi/concourse/commit/b0eb970b7902423283abccbe27dfeed394cd05e2

One caveat of this process is that we were not experts in all of the 23.000 projects represented, and therefore could only validate obvious mistakes. However, in this process we found no clear indications of mislabeled examples, meaning that all 200 negative examples we checked did seem to be consistent in practice. Therefore we are confident that our mining process produced a comprehensive dataset that mitigated this type of error.

6.2 Model Discussion

6.2.1 Replication

The previous findings of Panthaplackel et al.[28] and Steiner et al.[39] were replicated in our project. Thanks to the provided replication packages, we were not only able to reproduce the previous results, but we were also able to apply the previous models to our new dataset. However, we also found that the high performance of the previous models relied upon a data collection artifact that added spurious new lines to all positive examples, as shown in Figure 5.3. Because this artifact was not present in our dataset, we were unable to replicate the high performance of the previous models when we applied them to the new languages. Therefore, it appears that the previously reported high performance was predominantly due to these artifacts, casting doubts on the true generalizability and efficacy of the earlier models.

This result highlights the importance of trying to understand and interpret the results of a deep learning model, which may have picked up on unexpected artifacts in the data. For example, Lehman et al. found that a mushroom edibility classification model learned that positive and negative examples were presented in alternating order, and therefore learned to classify mushrooms based on their position in the dataset[18] rather than on their actual appearance. The tremendous capability of large models combined with their inscrutable complexity means that they often operate as black boxes, producing results without a clear, interpretable rationale. This combination lets them very quickly learn to make predictions based on any discernible patterns, including those that might be completely unrelated to the true underlying relationships in the data. It underscores the vulnerability of such models to what is often termed *shortcut learning*, where the model identifies and leverages shortcuts in the data to achieve high performance on the training set, but often at the expense of true generalizability.[12] As researchers, we have to exercise extra caution with collecting data in a way that prevents inadvertent patterns or artifacts from being introduced.

6.2.2 Explanations of Low Performance

Overall, even the highest overall weighted F1 score of 0.242 of CodeBERT on Java leaves significant room for improvement. In the setting of code comment consistency analysis, it is reasonable to discuss whether better performance is possible given the current methods. During the dataset validation phase section 6.1, we confirmed that we did not include false negatives in the dataset. However, we did take note of other potential issues in the dataset.

Missing Context For example, we noted that 44% of the Java methods we mined contained only one statement, calling out to another method. Without including the content of the referenced method in the context given to the model, in some of these cases it is difficult to imagine how the model could make the correct classification. Languages other than Java exhibit cases with this potential problem as well. Listing 6.3 gives an example from the newly mined training dataset of a Python comment we label as inconsistent because both the comment and code were changed in a single commit. In Listing 6.4, the example is shown after the change, where the change pertains entirely to context outside of the function body. In such a case, the model has no way of knowing why the comment is inconsistent with the code unless it has already encountered the entire sample during pre-training. However, we note that this issue appeared to be infrequent. After analyzing a random sample of 50 examples from the Python training set, we only found this one case where the outcome of classification was likely ambiguous given only the provided context.

```
# Comment: This must be a class method so
    a model may have properties that are
    of type self, this ensures that we
    don't create a cyclic import
def openapi_types():
    return {
        'value': ([float],),  # noqa:
            E501
    }
```

Listing 6.3: Python Example (Before)

```
# Comment: This must be a method because
    a model may have properties that are
    of type self, this must run after the
    class is loaded
def openapi_types():
    return {
        'value': ([float],),
    }
```

Listing 6.4: Python Example (After)

Inconsistent Standards We also encounter the fundamental issue of subjectivity of consistency classification in the dataset. Because we allow consistency to be implicitly defined by the patterns of contributions over thousands of projects, we will naturally encounter conflicts in the data. What one reviewer of pull requests would consider consistent might not pass the review process for another individual. Listing 6.5 gives an example from the training set of the Go language where the inconsistent labeled comment is most likely arguably consistent. The motivation for collecting a large dataset for this task is that over a large number of individuals, these clashes will resolve themselves into a working model of consistency that can be applied to as many instances as possible. However, if these conflicts occur too frequently, the performance of the model will be limited as its updates are pulled in conflicting directions.

```
// Comment: GetIdOk returns a tuple with
    the Id field value if set, nil
    otherwise and a boolean to check if
    the value has been set.
func (o *VerificationFlow) GetIdOk() (*
    string, bool) {
    if o == nil || o.Id == nil {
        return nil, false
    }
    return o.Id, true
}
```

Listing 6.5: Go Example (Before)

```
// Comment: GetIdOk returns a tuple with
    the Id field value and a boolean to
    check if the value has been set.
func (o *VerificationFlow) GetIdOk() (*
    string, bool) {
    if o == nil {
        return nil, false
    }
    return &o.Id, true
}
```

Listing 6.6: Go Example (After)

6.2.3 Negative Results in Context

Overall, the results of all tested models presented in Table 5.7 performed worse than initially expected. In Figure 5.3 and Figure 5.4, we demonstrated how a labeling artifact of trailing newlines allowed models to circumvent the test objective and achieve very good performance. Building upon this initial impression, we found out that in this setting the true results of the tested models was much worse in comparison. However, we believe that this result is still a valuable contribution to the field. Menzies et al. and González-Barahona et al. have written about a lack of reproduction studies in software engineering research, citing conclusion instability and difficulty in collecting new data following the original methodology.[23][13] In other words, the conclusions of one study might apply only to a narrow subset of the problem domain, and when tested in other ways no longer reproduces. Our work highlights that rigorous testing is needed in research by comparing both old and new datasets, and we support the application of understandability techniques to diagnose how deep learning models make their predictions.

The results we present in this thesis, while negative, are only an initial step towards further progress in the study of this problem. Because we committed resources to expanding the scope of the study to additional languages, we devoted less time to engineering model architectures and training methods to improve performance. Therefore, we hope that this work provides a motivation for future researchers to continue development in the setting of source code comment consistency prediction.

6.3 Benchmark Findings

We presented the results of our benchmark set in section 5.3. In this discussion we discuss implications of the different results between the benchmark set and mined dataset. We also share some of the challenges in creating this benchmark set which could be valuable to future researchers studying GitHub pull requests.

6.3.1 Validity of Results

Table 5.9 showed a significant decrease in performance from the automatically mined test set to the manually curated benchmark examples. One explanation of this result is that the construction of the benchmark set makes the differentiation between inconsistent and consistent examples more challenging. Because examples in the mined dataset are entire commits, in most cases the functionality in the code actually changes as well as the contents of the comment. However, in this pull request comment benchmark, almost all examples have code which does not change between the negative and positive label. Instead, only the comment is changed. Therefore, this test evaluates the model's ability to make a distinction between very similar inputs, which does not perfectly align with the training situation, where this was not presented.

6.3.2 Generality

In this section we briefly discuss whether the results on the benchmark set cast doubt on the modelling methodology. It is natural to expect the overall benchmark results to be lower than the training evaluation. After all, the pull request has already passed the author's own self-review, and comments made to it are an additional round of review.

This aspect is one place where the difference between industry and open source likely plays a part. If code is committed without peer review, then the training dataset, mined from commits, would only include changes that the original code author could notice in the first place. Although the filtering on projects included in the dataset precludes most low-quality code, the standard for strictness in code review in open source is likely lower overall than in large companies, where every commit must pass at least one expert peer review. Therefore, if we could guarantee that all commits included in the training set were reviewed by a human, as they are in industry, then the training and benchmarking objectives would align more closely. However, in open source data, without this guarantee, we could expect to see the results we observed, where the difficulty of the benchmark exceeded that of most of the training data. In Listing 6.7, we include an example of a difficult case from the benchmark set, where the code is the same between the old and new versions and most of the comment is shared as well.

```
/**
 * Sets whether or not to disconnect the
      client on detecting a channel error.
       By default, it is enabled and
 * client disconnects immediately. If it
      is disabled the client tries to
      reconnect according to {@link #
      maxReconnections(int)}.
 */
public EventStoreBuilder
    disconnectOnTcpChannelError(boolean
    disconnectOnTcpChannelError) {
    settingsBuilder.
        disconnectOnTcpChannelError(
        disconnectOnTcpChannelError);
    return this;
}
```

Listing 6.7: Java Benchmark Example (Before)

```
/**
 * Sets whether or not to disconnect the
      client on detecting a channel error.
       By default, it is disabled and the
      client
 * tries to reconnect according to {@link
      #maxReconnections(int)}. If it is
      enabled the client disconnects
      immediately.
 */
public EventStoreBuilder
    disconnectOnTcpChannelError(boolean
    disconnectOnTcpChannelError) {
    settingsBuilder.
        disconnectOnTcpChannelError(
        disconnectOnTcpChannelError);
    return this;
}
```

Listing 6.8: Java Benchmark Example (After)

6.3.3 Challenges

In this section we list some of the challenges we encountered in creating the benchmark set. We share these challenges because they could be relevant to future work that involves study of pull requests. To our knowledge, our dataset of over 13 million pull request comments, which we used to build our benchmark set, is one of the largest of its kind, and we hope that it can be used for future research.

- *"Comment" overload*: Clearly, there is a name clash between the topic of our study and the context within a GitHub pull request. Reviewers make "comments" on the content of a pull request, and we are specifically interested in "comments" about "comments". For example, when we filter for review text containing "this comment", most of the matches we found were referring to other review comments, not code comments.

- *Ignored change requests*: A surprisingly large number of instances where reviewers requested a change to a comment were simply ignored, meaning that the original code was merged, without fixing a mistake in a comment[2]. We might surmise that this behavior comes from a lower standard of review in the open source world than in industry.

- *Squashed commits*: A common practice in managing multiple branches of a distributed version control system is to squash multiple commits into a single commit upon merge into the main branch. This step ensures that the history of the main branch is linear and easier to work with. However, this squash step removes all of the intermediate commits of the original pull request. In our study, we are specifically interested in these commits, where an author responds to a code review by committing a change to their original submission. We found that most examples of updated code were lost in this squash operation.[3]

The combination of these challenges greatly increased the difficulty and time required to create the benchmark sets for each language.

6.4 RQ4: Social Study

While we found that almost all of our proposed fixes were accepted (18 out of 20), we received no responses on our survey. The pull request results are listed in Table 5.10. Given that the changes were generally accepted, we can conclude that these changes were indeed useful to the maintainers. Even though the sample size of 20 is not extremely comprehensive, the overwhelming majority of requests being accepted means that the conclusion is not in doubt. However, without more qualitative reports from the survey, it is difficult to place these findings in the broader context of software engineering development tooling research. We can point to several factors that likely influenced the response rate on our survey questions.

- *Survey Targeting*: Because the survey link was given generically in the body of the pull request description, our reviewers may have overlooked it or regarded it as non-essential. Past work in developer surveys has indicated a better response rate from surveys which are personalized and sent to individual contributors.[45][34]

- *Introduction Hurdle*: If a reviewer did click our survey form, they would have been greeted with a 9 paragraph privacy agreement they must consent to. For an unfamiliar and busy open source software maintainer, that might have seemed overwhelming, and they may have responded by leaving our survey.

- *Nature of Pull Requests*: The pull requests sent as part of this research were relatively simple, as shown in the example of Figure 5.7. Therefore, reviewers may have felt that they had no additional insight to give because the acceptance of the pull request was evidently self-explanatory.

[2] For example, `https://github.com/openshift/openshift-azure/pull/238#discussion_r212902362`
[3] For example, `https://github.com/proteneer/timemachine/pull/640#discussion_r808454189`

Overall, we argue that our results from the acceptance rates of pull requests are still valid, despite the lack of survey responses. From the 20 pull requests submitted, 18 were accepted, a rate of 90%. Given the sample size of 20, with a 95% confidence interval, the true acceptance rate is between 76% and 100%. In active open source projects, a high volume of issues and pull requests can overwhelm maintainers, so they prioritize pull requests that provide clear benefits without a heavy burden. The fact that 18 out of 20 requests were accepted implies that fixing the issue of comment consistency has a positive value. Due to this high acceptance rate, we can conclude that pull requests that fix comment code correspondences are useful to project maintainers.

Chapter 7

Conclusions

In this chapter, we summarize our major findings and reflect on the overall impact of this project.

In this thesis, we have explored the intersection between code-focused large language models and the study of code and comment consistency. The study of code comment consistency involves the classification of comments as either consistent or inconsistent with the code they are attached to. In this work, we focused on method or function level comments, which are linked to a specific method or function in the code. We addressed a major gap in previous work[30] by evaluating the established approach on languages other than Java. This work involved a large scale software mining operation on GitHub to build a balanced dataset with more than 80,000 examples. We also increased the scope of our study by including real-world evaluation with a manually curated benchmark set and validating the approach on current day open source projects through a search of more than 13 million pull request comments on GitHub.

In summary, we found that CodeBERT outperformed previous models in evaluation on both previous datasets and our new dataset. This result supports the theory that pretrained large language models can reduce the need for language-specific feature engineering. Furthermore, we identified a significant anomaly in an existing Java dataset that may have affected previously published results. This anomaly involved a whitespace artifact that models could use to distinguish positive and negative cases which was not present in the source data. All positive cases in the existing dataset ended with an extra pair of newline characters, which allowed a trivial classifier to achieve 100% precision and high recall. We also tested that our new dataset was not vulnerable to this issue, and therefore the resulting evaluation scores of all models were lower than previously observed.

In the real-world benchmark evaluation phase, we observed a gap between model evaluation performance and benchmark results. We hypothesize that this gap is due to the difficulty of the benchmark task, which requires the model to make a more fine-grained distinction between classes than it typically encountered in the training task. This work highlighted the challenge of real-world situations and the danger of assuming that model performance on a training task would necessarily translate to equivalent performance in operation.

Finally, we found that 18 out of 20 open source maintainers accepted our handwritten contributions that addressed comment issues. This finding gives additional evidence that the field of comment consistency classification is valuable and worth pursuing in tooling development.

We hope that these new results and the expanded datasets we publish alongside this thesis will catalyze further growth across this field. For example, our datasets of labeled code and comment pairs could help train comment generation models. Our dataset of over 13 million pull request comments can help researchers build pull request review bots, or aid with linguistic analysis in an open source software engineering setting.

7.1 Future Work

In this section we share the most impactful next steps for research in this field, in our estimation.

- *Comment Generation*: Although we excluded this direction of study because we identified code comprehension as the bottleneck of software development, this topic is very active. We believe that future work could build a comment generation solution by using the datasets we have collected, which have mined a large amount of open source data, in a format of comment code pairs, suitable for the training setting of comment generation.

- *Contrastive Training Techniques*: One point we made in our discussion of benchmark set results was that the setting was more challenging than the model encountered during training. Because differences before and after changes requested were minor, a successful model would need to be able to differentiate small details in the final outcome. One potential approach to achieving this capability is to incorporate a contrastive loss into the training procedure, so the model makes a stronger distinction between classes when the content is similar. Choi et al. have recently demonstrated the promise of this approach in the coding domain, and these results would likely be applicable in this setting as well.[4]

- *Broad Spectrum or Focused Training*: As discussed in chapter 6, different projects have different standards on what comment consistency entails. Future work might revisit the hypothesis that a model can improve from training on many projects as opposed to being targeted at a single project or organization. In other words, sacrificing some level of generality for higher performance on a small set of repositories could be achievable. While we showed that a model trained on all 4 programming languages scores more highly than a model trained on only one language, we did not test against a model that was only focused on a handful of code repositories.

- *Augmented Input Context*: A major shortcoming of our presented results is an overall less than stellar performance in both the automatically mined test set and in the manually curated benchmark set. We identified several likely causes in chapter 6, including a possible lack of adequate context for an informed decision. For example, 44% of all example methods from Java consisted of only a single statement. We believe that future work can look toward techniques to effectively expand the input with surrounding context that could help the model make a determination where the method body itself does not have enough information. For instance, the model could be given the entire surrounding class, or code file, or references to called subroutines could be inlined in the input to the model so their contents are also available for prediction.

- *Targeted Social Study*: One limitation in our project was the lack of response on our survey. Although we were still able to draw some conclusions about our work, more detailed qualitative opinions could clarify our results in the context of other research. Future work can likely gain more qualitative insights on code and comment consistency by targeting questions directly at developer groups or individuals. For example, Steinmacher et al. received 24 open-ended questionnaire responses from a survey sent to 19 open source project mailing lists.[40] In this case, we would like to judge the relative perceived importance of fixing comment consistency against other code comprehension issues such as excessive function redirection. It would also be valuable to fully test the hypothesis that developers indeed respond positively to a classification-only model that could identify potential mistakes without fixing them on its own.

Bibliography

[1] J Alammar. Ecco: An open source library for the explainability of transformer language models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, 2021. 24, 32

[2] Huanchao Chen, Yuan Huang, Zhiyong Liu, Xiangping Chen, Fan Zhou, and Xiaonan Luo. Automatically detecting the scopes of source code comments. *Journal of Systems and Software*, 153:45–63, 2019. 20

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. 14, 16, 28

[4] Seungtaek Choi, Myeongho Jeong, Hojae Han, and Seung-won Hwang. C2l: Causally contrastive learning for robust text classification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 10526–10534, 2022. 48

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. 13, 20

[6] Peter Elmers. Github public repository metadata. `https://www.kaggle.com/datasets/pelmers/github-repository-metadata-with-5-stars`. ix, 7, 8, 25

[7] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. The effect of poor source code lexicon and readability on developers' cognitive load. In *Proceedings of the 26th Conference on Program Comprehension*, pages 286–296, 2018. 2, 9

[8] Nils Feldhus, Leonhard Hennig, Maximilian Nasert, Christopher Ebert, Robert Schwarzenberg, and Sebastian Mller. Saliency map verbalization: Comparing feature importance representations from model-free and instruction-based methods. In *Proceedings of the 1st Workshop on Natural Language Reasoning and Structured Explanations (NLRSE)*, pages 30–46, 2023. ix, 24

[9] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020. ix, 14, 15, 21, 22, 36

[10] Beat Fluri, Michael Wursch, and Harald C Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 70–79. IEEE, 2007. 19

[11] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The Pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020. 15

[12] Robert Geirhos, Jörn-Henrik Jacobsen, Claudio Michaelis, Richard Zemel, Wieland Brendel, Matthias Bethge, and Felix A Wichmann. Shortcut learning in deep neural networks. *Nature Machine Intelligence*, 2(11):665–673, 2020. 42

[13] Jesús M González-Barahona and Gregorio Robles. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empirical Software Engineering*, 17:75–89, 2012. 43

[14] Miguel Hoyos Ruge et al. Analysis of software engineering automation tools for go. 2021. 8

[15] Shin Hwei Tan, Chunfeng Hu, Ziqiang Li, Xiaowen Zhang, and Ying Zhou. Github-oss fixit: Fixing bugs at scale in a software engineering course. *arXiv e-prints*, pages arXiv–2011, 2020. 17

[16] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101, 2014. xi, 15, 16, 25

[17] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020. 22

[18] Joel Lehman, Jeff Clune, and Dusan Misevic. The surprising creativity of digital evolution. In *Artificial Life Conference Proceedings*, pages 55–56. MIT Press One Rogers Street, Cambridge, MA 02142-1209, USA journals-info . . . , 2018. 42

[19] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019. 20

[20] Zhiyong Liu, Huanchao Chen, Xiangping Chen, Xiaonan Luo, and Fan Zhou. Automatic detection of outdated comments during code changes. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 154–163. IEEE, 2018. 11, 19

[21] Christopher Manning and Hinrich Schutze. *Foundations of statistical natural language processing*. MIT press, 1999. 13

[22] Diego Marcilio, Carlo A Furia, Rodrigo Bonifácio, and Gustavo Pinto. Automatically generating fix suggestions in response to static code analysis warnings. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 34–44. IEEE, 2019. 17

[23] Tim Menzies and Martin Shepperd. Special issue on repeatable results in software engineering prediction, 2012. 43

[24] Shervin Minaee, Nal Kalchbrenner, Erik Cambria, Narjes Nikzad, Meysam Chenaghlu, and Jianfeng Gao. Deep learning–based text classification: a comprehensive review. *ACM computing surveys (CSUR)*, 54(3):1–40, 2021. 2, 8

[25] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22:3219–3253, 2017. 25, 29

[26] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint*, 2022. ix, 15, 16, 20, 22, 28

[27] Stack Overflow. Stack Overflow developer survey 2020. 2

[28] Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J Mooney. Deep just-in-time inconsistency detection between comments and source code. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 427–435, 2021. ix, 6, 9, 11, 12, 19, 20, 21, 23, 26, 31, 32, 37, 42

[29] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 13

[30] Pooja Rani, Arianna Blasi, Nataliia Stulova, Sebastiano Panichella, Alessandra Gorla, and Oscar Nierstrasz. A decade of code comment quality assessment: A systematic literature review. *Journal of Systems and Software*, page 111515, 2022. 2, 6, 8, 9, 11, 12, 47

[31] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4):58–66, 2018. 24

[32] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Automatically assessing code understandability: How far are we? In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 417–427, 2017. 12

[33] Simone Scalabrino, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Improving code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10, 2016. 12

[34] Edward Smith, Robert Loftin, Emerson Murphy-Hill, Christian Bird, and Thomas Zimmermann. Improving developer participation rates in surveys. In *2013 6th International workshop on cooperative and human aspects of software engineering (CHASE)*, pages 89–92. IEEE, 2013. 45

[35] Xiaotao Song, Hailong Sun, Xu Wang, and Jiafei Yan. A survey of automatic generation of source code comments: Algorithms and techniques. *IEEE Access*, 7:111411–111428, 2019. 13

[36] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. A human study of comprehension and code summarization. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 2–13, 2020. 13

[37] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. Quality analysis of source code comments. In *2013 21st international conference on program comprehension (icpc)*, pages 83–92. Ieee, 2013. 2, 5, 11

[38] Marcel Steinbeck and Rainer Koschke. Javadoc violations and their evolution in open-source software. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 249–259, 2021. 7

[39] Theo Steiner and Rui Zhang. Code comment inconsistency detection with bert and longformer. *arXiv preprint arXiv:2207.14444*, 2022. 9, 13, 23, 32, 37, 42

[40] Igor Steinmacher, Marco Gerosa, Tayana U Conte, and David F Redmiles. Overcoming social barriers when contributing to open source software projects. *Computer Supported Cooperative Work (CSCW)*, 28:247–290, 2019. 48

[41] Klaas-Jan Stol and Brian Fitzgerald. The abc of software engineering research. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 27(3):1–51, 2018. 29

[42] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T Leavens. @ tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 260–269. IEEE, 2012. 29

[43] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017. 14

[44] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. A large-scale empirical study on code-comment inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 53–64. IEEE, 2019. 6, 16, 28

[45] Mairieli Wessel, Alexander Serebrenik, Igor Wiese, Igor Steinmacher, and Marco A Gerosa. What to expect from code review bots on github? a survey with oss maintainers. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, pages 457–462, 2020. 45

[46] Xiaoya Xia, Shengyu Zhao, Xinran Zhang, Zehua Lou, Wei Wang, and Fenglin Bi. Understanding the archived projects on github. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 13–24. IEEE, 2023. 30

[47] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2017. 29

[48] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2018. 2, 5

[49] Bai Yang, Zhang Liping, and Zhao Fengrong. A survey on research of code comment. In *Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences*, pages 45–51, 2019. 2, 6

[50] Fiorella Zampetti, Cedric Noiseux, Giuliano Antoniol, Foutse Khomh, and Massimiliano Di Penta. Recommending when design technical debt should be self-admitted. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 216–226. IEEE, 2017. 20

[51] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023. 21

[52] Junji Zhi, Vahid Garousi-Yusifoğlu, Bo Sun, Golara Garousi, Shawn Shahnewaz, and Guenther Ruhe. Cost, benefits and quality of software development documentation: A systematic mapping. *Journal of Systems and Software*, 99:175–198, 2015. 11

Appendix

A.1 Data Filtering Examples

In this appendix we include some examples of potential functions that were filtered out of our dataset. The reason for filtering is described in each code block's caption.

```go
//go:build !ignore_autogenerated
// +build !ignore_autogenerated

// Code generated by controller-gen. DO NOT EDIT.

package v1beta1

import (
  "k8s.io/api/core/v1"
  runtime "k8s.io/apimachinery/pkg/runtime"
)

// DeepCopyInto is an autogenerated deepcopy function, copying the receiver, writing into out. in must be non-nil.
func (in *AdminServerServicePolicy) DeepCopyInto(out *AdminServerServicePolicy) {
  *out = *in
  if in.Annotations != nil {
    in, out := &in.Annotations, &out.Annotations
    *out = make(map[string]string, len(*in))
    for key, val := range *in {
      (*out)[key] = val
    }
  }
}
```

Listing A.1: Generated pravega/zookeeper-operator: api/v1beta1/zz_generated.deepcopy.go

```go
// Code generated by github.com/Khan/genqlient, DO NOT EDIT.

package main

import (
  "context"
  "time"

  "github.com/Khan/genqlient/graphql"
)

// __getUserInput is used internally by genqlient
type __getUserInput struct {
  Login string `json:"Login"`
}

// GetUser returns getUserResponse.User, and is useful for accessing the field via an interface.
func (v *getUserResponse) GetUser() getUserUser { return v.User }
```

Listing A.2: Generated Khan/genqlient: example/generated.go

```
/**
 * Move the popover to the |placement| position of the object located on the |rect|.
 *
 * @param popover {Object} The popover object to be moved.
 * @param placement {String} The relative position to move the popover — top | bottom | left | right.
 * @param align {String} The way the popover should be aligned — center | left | right.
 * @param rect {ClientRect} The ClientRect of the object to move the popover around.
 * @param triangle {Object} The element that contains the popover's triangle. This can be null.
 */
function move(popover, placement, align, rect, triangle) {
  var containerRect;
  var popoverRect = getBoundingClientRect(popover[0]);
  var popoverRight;
  var top, left;
  // ...
}
```

Listing A.3: Duplicated, nohros/nsPopover: src/nsPopover.js and nohros/nsPopover: example/nsPopover.js

```
// TODO: clean up after PR#138 is merged and tested https://github.com/GoogleCloudPlatform/gcping/pull/138
// EndpointsFromServer is used by the cli to generate an Endpoint map
// using json served by the gcping endpoints.
func EndpointsFromServer(ctx context.Context, endpointsURL string) (map[string]Endpoint, error) {
  req, err := http.NewRequestWithContext(
    ctx,
    http.MethodGet,
    endpointsURL,
    nil,
  )
  if err != nil {
    return nil, err
  }
  resp, err := http.DefaultClient.Do(req)
  if err != nil {
    return nil, err
  }
  defer resp.Body.Close()
  if resp.StatusCode != http.StatusOK {
    return nil, fmt.Errorf("%v %s", resp.Status, endpointsURL)
  }
  e := make(map[string]Endpoint)
  decoder := json.NewDecoder(resp.Body)
  if err := decoder.Decode(&e); err != nil {
    return nil, err
  }
  return e, err
}
```

Listing A.4: Code with technical debt comment GoogleCloudPlatform/gcping: internal/config/endpoints.go

```
// DialSlashGraphQLEndpoint is deprecated and will be removed in v21.07 release. For more details,
// see: https://discuss.dgraph.io/t/regarding-slash-cloud-dgraph-endpoints-in-the-clients/13492
// DialSlashGraphQLEndpoint is deprecated, as it leaks GRPC connections.
// Please use DialSlashEndpoint instead
func DialSlashGraphQLEndpoint(endpoint, key string) (*Dgraph, error) {
  conn, err := DialSlashEndpoint(endpoint, key)
```

```
7     if err != nil {
        return nil, err
      }
      dc := api.NewDgraphClient(conn)
      dg := NewDgraphClient(dc)
12    return dg, nil
}
```

Listing A.5: Deprecated dgraph-io/dgo: client.go

A.2 Submitted Pull Requests

1. https://github.com/hmcts/civil-service/pull/2869
2. https://github.com/ansys/pyaedt/pull/3128
3. https://github.com/chaoss/augur/pull/2442
4. https://github.com/Dallinger/Dallinger/pull/5293
5. https://github.com/demisto/content/pull/27489
6. https://github.com/Flexget/Flexget/pull/3796
7. https://github.com/gdsfactory/gdsfactory/pull/1790
8. https://github.com/cupy/cupy/pull/7655
9. https://github.com/DLR-RM/stable-baselines3/pull/1567
10. https://github.com/CircuitVerse/CircuitVerse/pull/3834
11. https://github.com/salesforce/lwc/pull/3593
12. https://github.com/benthosdev/benthos/pull/1997
13. https://github.com/Praqma/helmsman/pull/807
14. https://github.com/spacemeshos/go-spacemesh/pull/4666
15. https://github.com/Versent/saml2aws/pull/1089
16. https://github.com/go-gorm/gen/pull/906
17. https://github.com/deso-protocol/core/pull/574
18. https://github.com/chanzuckerberg/fogg/pull/889
19. https://github.com/sensu/sensu-go/pull/5027
20. https://github.com/zalando-incubator/kubernetes-on-aws/pull/6100